

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/345985082>

Introdução à Linguagem R: seus fundamentos e sua prática

Book · June 2021

CITATIONS

2

READS

5,197

2 authors:



[Pedro Faria](#)

Universidade Federal de Ouro Preto

1 PUBLICATION 2 CITATIONS

[SEE PROFILE](#)



[Joao Parga](#)

Instituto de Pesquisa Econômica Aplicada - IPEA

7 PUBLICATIONS 37 CITATIONS

[SEE PROFILE](#)

INTRODUÇÃO À LINGUAGEM R

Seus fundamentos e sua prática

Segunda edição

Pedro Duarte Faria

Take Blip

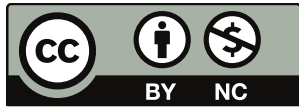
João Pedro Figueira Amorim Parga

Instituto de Pesquisa Econômica Aplicada (IPEA)

Belo Horizonte

Junho de 2021

Copyright © 2021, 2020 by Pedro Duarte Faria and João Pedro Figueira Amorim Parga.



Esta obra está licenciada com uma Licença Creative Commons - Atribuição - NãoComercial 4.0 Internacional. Para ver uma cópia dessa licença, visite o endereço: <http://creativecommons.org/licenses/by-nc/4.0/>.

Você é livre para compartilhar, redistribuir, transformar ou adaptar essa obra, desde que você não venha a utilizá-la em nenhuma atividade de propósito comercial, e apenas enquanto a atribuição é dada aos autores dessa obra.

ISBN (Digital): 978-65-00-12606-8

Como citar essa obra:

FARIA, Pedro Duarte; PARGA, João Pedro Figueira Amorim. *Introdução à Linguagem R: seus fundamentos e sua prática*. 2. ed. Belo Horizonte: [s.n.], 2021. ISBN 978-65-00-12606-8.

Disponível em: https://pedro-faria.netlify.app/pt/publication/book/introducao_linguagem_r/

Autor correspondente e mantenedor da obra:

Pedro Duarte Faria

Contato: pedropark99@gmail.com

Site pessoal: <https://pedro-faria.netlify.app/>

Onde encontrar esse livro:

A primeira edição deste livro foi publicada no dia 10 de Novembro de 2020, e desde então, passou por diversas revisões e expansões. Você sempre pode encontrar a versão mais atualizada deste trabalho, no site pessoal de Pedro Duarte Faria.

Última atualização: 27 de junho de 2021.

Publicação do livro:

[<https://pedro-faria.netlify.app/pt/publication/book/introducao_linguagem_r/>](https://pedro-faria.netlify.app/pt/publication/book/introducao_linguagem_r/)

Visite o projeto:

Este livro faz parte de um projeto pessoal de Pedro Duarte Faria. O projeto tem como objetivo, criar e compartilhar conhecimento sobre a linguagem R, em especial com a comunidade brasileira, que ainda carece de materiais amplos e que abordam os fundamentos da linguagem.

Projeto:

[<https://pedro-faria.netlify.app/pt/project/r_curso/>](https://pedro-faria.netlify.app/pt/project/r_curso/)

Sumário

Sobre os autores	1
Prefácio	3
O que é esse livro?	3
Novidades desta segunda edição e o futuro desta obra	5
Porque aprender R? Quais são as suas vantagens?	5
1 Noções Básicas do R	11
1.1 Uma descrição do R	11
1.2 Introdução ao R e RStudio: noções básicas	13
1.3 Introdução a objetos	18
1.4 Funções (noções básicas)	22
1.5 Erros e ajuda: como e onde obter	24
1.6 Scripts	32
1.7 Pacotes	37
2 Fundamentos da Linguagem R	43
2.1 Introdução	43
2.2 Objetos (uma revisão)	43
2.3 Estruturas e tipos de dados	47
2.4 Estruturas de dados	49
2.5 Tipos de dados	68
2.6 Coerção no R	72
2.7 <i>Subsetting</i>	74
2.8 Valores especiais do R	84
3 Importando e exportando dados com o R	93
3.1 Introdução e pré-requisitos	93
3.2 Fontes de dados	93
3.3 Diretório de trabalho	94
3.4 Definindo endereços do disco rígido no R	95
3.5 Plataforma de Projetos do RStudio	98

3.6	Importando arquivos de texto com <code>readr</code>	101
3.7	Um estudo de caso: lendo os microdados da PNAD Contínua com <code>read_fwf()</code> . . .	111
3.8	Exportando os seus dados com o pacote <code>readr</code>	125
3.9	Importando planilhas do Excel com <code>readxl</code>	126
3.10	Importando arquivos do SPSS, Stata e SAS com o pacote <code>haven</code>	131
3.11	<i>Encoding</i> de caracteres	135
4	Transformando dados com <code>dplyr</code>	145
4.1	Introdução e pré-requisitos	145
4.2	Panorama e padrões do pacote <code>dplyr</code>	145
4.3	Operador <i>pipe</i> (<code>%>%</code>)	146
4.4	Selecionando colunas com <code>select()</code>	150
4.5	Filtrando linhas com <code>filter()</code>	157
4.6	Ordenando linhas com <code>arrange()</code>	167
4.7	Adicionando variáveis à sua tabela com <code>mutate()</code>	169
4.8	Agrupando dados e gerando estatísticas sumárias com <code>group_by()</code> e <code>summarise()</code> .	174
4.9	A função <code>across()</code> como a grande novidade	181
4.10	Removendo duplicatas com <code>distinct()</code>	184
4.11	Combinando tabelas com <code>bind_cols()</code> e <code>bind_rows()</code>	186
5	Funções e Loops no R	195
5.1	Introdução	195
5.2	Noções básicas de <i>environments</i>	195
5.3	Uma introdução teórica às funções no R	202
5.4	Construindo um conjunto de funções	206
5.5	Introduzindo loops	211
5.6	Um estudo de caso: uma demanda real sobre a distribuição de ICMS	215
6	Introdução a base de dados relacionais no R	225
6.1	Introdução e pré-requisitos	225
6.2	Dados relacionais e o conceito de <i>key</i>	225
6.3	Introduzindo <i>joins</i>	229
6.4	Configurações sobre as colunas e <i>keys</i> utilizadas no <i>join</i>	230
6.5	Diferentes tipos de <i>join</i>	236
6.6	Relações entre <i>keys</i> : <i>primary keys</i> são menos comuns do que você pensa	240
7	<i>Tidy Data</i>: Uma abordagem para organizar os seus dados	247
7.1	Introdução e pré-requisitos	247
7.2	O que é <i>tidy data</i> ?	247
7.3	Operações de pivô	253
7.4	Completando e expandindo a sua tabela	270
7.5	Preenchendo valores não-disponíveis (NA)	279
7.6	Um estudo de caso sobre médias móveis com <code>complete()</code> e <code>fill()</code>	286

8	Visualização de dados com ggplot2	301
8.1	Introdução e pré-requisitos	301
8.2	O que é o ggplot e a sua gramática	301
8.3	Iniciando um gráfico do ggplot	304
8.4	Uma outra forma de se compreender o <i>aesthetic mapping</i>	314
8.5	Sobrepondo o <i>aesthetic mapping</i> inicial em diversas camadas	319
8.6	Uma discussão sobre os principais geom's	323
8.7	Exportando os seus gráficos do ggplot	352
9	Configurando componentes estéticos do gráfico no ggplot2	371
9.1	Introdução e pré-requisitos	371
9.2	Tema (<i>theme</i>) do gráfico	371
9.3	Eliminando elementos do gráfico	373
9.4	Alterando a temática de textos	374
9.5	Plano de fundo (<i>background</i>) e <i>grid</i>	378
9.6	Eixos do gráfico	381
9.7	Configurações temáticas em uma legenda	383
9.8	Alterando a temática em facetas	388
9.9	Alterando as fontes do seu gráfico	389
10	Manipulação e transformação de <i>strings</i> com stringr	399
10.1	Introdução e pré-requisitos	399
10.2	Algumas noções básicas	399
10.3	Concatenando ou combinando <i>strings</i> com <code>paste()</code> e <code>str_c()</code>	401
10.4	Vantagens do pacote stringr	407
10.5	Comprimento de <i>strings</i> com <code>str_length()</code>	407
10.6	Lidando com capitalização e espaços em branco	408
10.7	Extraindo partes ou <i>subsets</i> de um <i>string</i> com <code>str_sub()</code>	410
10.8	Expressões regulares (ou <i>regex</i>) com <code>str_detect()</code>	414
10.9	Substituindo partes de um texto com <code>str_replace()</code>	437
10.10	Dividindo <i>strings</i> com <code>str_split()</code>	439
10.11	Extraindo apenas a correspondência de sua expressão regular com <code>str_extract()</code>	440
11	Introduzindo fatores (<i>factor</i>'s) com forcats	447
11.1	Introdução e pré-requisitos	447
11.2	O que são fatores ?	447
11.3	Como construir um fator	448
11.4	Porque utilizar fatores se eu posso armazenar como texto ?	450
11.5	Não construir o atributo <code>levels</code> é contraintuitivo	451
11.6	Alterando a ordem dos níveis de um fator	452
11.7	Reordenando fatores em gráficos	455
11.8	Modificando os níveis de um fator	460

12 Introdução à variáveis de tempo com lubridate	463
12.1 Introdução e pré-requisitos	463
12.2 O pacote lubridate	463
12.3 Datas com o tipo Date	464
12.4 Datas, horários e fusos horários com POSIXct e POSIXlt	470
12.5 Diferenças entre POSIXct e POSIXlt	481
12.6 Extraindo os componentes de uma variável de tempo	484
12.7 Fusos horários	487
12.8 Calculando intervalos com o tipo difftime	495
12.9 Como as variáveis de tempo são interpretadas pelo R ?	497
 Respostas dos exercícios de cada capítulo	 505
 A PNAD Contínua: arquivo CSV para input	 573

Sobre os autores

Pedro Duarte Faria

Pedro Duarte Faria é graduando em Economia pela Universidade Federal de Ouro Preto - UFOP. Atualmente é Analista de Performance e Inteligência de Negócios na Take Blip. Como pesquisador, tem atuado em especial na área de Economia da Ciência e da Tecnologia, tendo ganhado recentemente um prêmio por sua pesquisa apresentada no XXI Seminário de Economia Industrial (SEI), realizado pelo GEEIN/FCLAr-UNESP.

Lattes: <<http://lattes.cnpq.br/0308632529554550>>

Site pessoal: <<https://pedro-faria.netlify.app/>>

João Pedro Figueira Amorim Parga

João Pedro Figueira Amorim Parga é mestre em Economia pelo CEDEPLAR-UFMG (2020), e possui graduação em Economia pela mesma instituição. Atualmente é Pesquisador Assistente no Instituto de Pesquisa Econômica Aplicada (IPEA). Possui experiência em Economia Regional e Urbana, especialmente nos seguintes temas: distribuição espacial de atividades econômicas, setor de serviços, ciência de dados, habitação, aglomeração espacial e geografia econômica.

Lattes: <<http://lattes.cnpq.br/8639351648030747>>

Prefácio

O que é esse livro?

Este livro surgiu inicialmente, como um material de apoio aos pesquisadores e alunos do Curso Introdutório de R, que foi realizado durante o primeiro semestre de 2020, na Fundação João Pinheiro¹ (FJP-MG). O projeto foi idealizado na época, por um conjunto de três pessoas, dentre elas, estão os autores desta obra: Pedro Duarte Faria e João Pedro Figueira Amorim Parga. Portanto, esse material é resultado dessa experiência de ensino, onde buscamos compartilhar conhecimentos sobre essa linguagem com outras pessoas. Eu como professor, aluno e economista, sou muito grato por ter compartilhado essas experiências, com meu querido colega João Pedro Figueira Amorim Parga, que me ajudou a montar esse livro.

As origens da linguagem R, remetem a um dos mais importantes laboratórios de pesquisa do mundo, a Bell Labs, localizada nos EUA. Por sua origem, a enorme maioria dos materiais de referência a respeito da linguagem, estão em inglês, incluindo as principais fontes de ajuda da linguagem, como o [StackOverflow](#), ou as páginas e manuais internos do [CRAN R](#).

Entretanto, a comunidade de R no Brasil, tem se expandido constantemente nos últimos anos. Brasileiros tem desenvolvido importantes pacotes para a linguagem, que trazem grande apoio à produção científica do país. Apenas para citar alguns desses excelentes trabalhos, estão [Pereira et al. \(2020\)](#), [Petruszalek \(2016\)](#), [McDonnell, Oliveira e Giannotti \(2020\)](#), [Siqueira \(2020\)](#), [Braga, Assuncao e Hidalgo \(2020\)](#). Como resultado, bons materiais em português, de referência e apoio à linguagem tem surgido. Exemplos são: os materiais curtos montados pelo [Curso R](#); os trabalhos realizados pelos capítulos brasileiros do grupo [R-Ladies](#), como os [posts do capítulo de Belo Horizonte](#), e os [encontros desenvolvidos pelo capítulo de São Paulo](#); além de alguns materiais produzidos pelo Departamento de Estatística da UFPR, como um [site de apoio ao seu curso](#), ou este produzido por um dos professores do departamento, o [Dr. Walmes Marques Zeviani](#).

Porém, mesmo com esse avanço, grande parte desses conteúdos em português geralmente caem em algum desses dois problemas: 1) carecem de profundidade, ou de detalhamento sobre o que está “ocorrendo nos bastidores”. Em outras palavras, esses materiais são um pouco abstratos, pois

¹A Fundação João Pinheiro (fundada em 1969), é uma instituição de pesquisa e ensino vinculada à Secretaria de Estado de Planejamento e Gestão de Minas Gerais, e é responsável por produzir as principais estatísticas econômicas, sociais e demográficas do estado de Minas Gerais.

tentam abordar muita coisa em um espaço muito curto, sem dar o devido tempo a cada um dos componentes por trás da linguagem; 2) ou são especializados demais. Por exemplo, materiais que ensinam como estimar modelos específicos (ex: regressão linear sobre dados em painel), ou a trabalhar com bases de dados específicas (ex: PNAD contínua). Em outras palavras, esses materiais concedem em geral, uma visão muito restrita sobre a linguagem, e que é de difícil transposição para outros cenários e necessidades práticas.

Esses problemas emergem do próprio objetivo que esses materiais buscam cumprir. Como exemplo, os materiais escritos pelo [Curso-R](#) carregam certa abstração, pois em nenhum momento esses materiais pretendem oferecer uma revisão completa e profunda sobre o tema, mas sim, tutoriais rápidos e úteis, que lhe mostram o básico. Tendo isso em mente, esta obra em específico, representa a nossa tentativa de combater esses dois problemas. Ao discutir pacotes largamente utilizados nas mais diversas aplicações, além de fornecer uma visão aprofundada sobre os fundamentos (ou a teoria) da linguagem R. Por isso, o público-alvo deste livro são os brasileiros que desejam obter uma base mais sólida e uma visão mais abrangente da linguagem, de forma que eles possam identificar mais facilmente, as possíveis soluções que o R oferece para vários problemas diferentes de seu trabalho.

Por essas razões, este material é até certo ponto, prolixo em muitos assuntos aos quais são comumente tratados como simples e rápidos de se compreender (e.g. Objetos). Ao mesmo tempo, este material certamente busca ser descritivo, e não poupa detalhes em assuntos que são complexos e de difícil compreensão (e.g. Funções e *Loops*). Para mais, fornecemos ao longo da obra, diversos diagramas e representações visuais, que ajudam o leitor, a formar um modelo mental sobre como a linguagem R funciona. A obra também oferece exercícios ao final de cada capítulo, além de suas respostas ao final do livro.

Vários exemplos são fornecidos em cada tópico. Alguns desses exemplos são reais e retirados diretamente de nosso dia-a-dia com a linguagem. Já uma outra parte desses exemplos, buscam evidenciar ou demonstrar problemas práticos que podem emergir de seu trabalho com a linguagem e, portanto, mostrar quais são as possíveis soluções a serem empregadas. Dessa forma, podemos construir um *workflow*, ou um modelo mental de trabalho com a linguagem, ao longo de diferentes tópicos importantes para a sua aplicação prática em análise de dados.

Você sempre pode encontrar uma versão atualizada dessa obra, em sua [página de publicação](#). Este documento foi criado dentro do RStudio, por meio do pacote `rmarkdown` e do sistema $\text{\LaTeX}$. Grande parte do conhecimento exposto aqui, está baseado em diversas referências sobre a linguagem R, em especial os trabalhos de [Wickham \(2015a\)](#), [Peng \(2015\)](#), [Wickham e Grolemund \(2017\)](#), [Long e Teetor \(2019\)](#), assim como a documentação oficial da linguagem R ([R CORE TEAM, 2020b](#); [R CORE TEAM, 2020a](#)).

Pedro Duarte Faria
06/06/2021
Belo Horizonte - MG
Brasil

Novidades desta segunda edição e o futuro desta obra

Esta segunda edição traz dois importantes complementos que aproximam esta obra de seu objetivo principal (ser um livro técnico e introdutório sobre a linguagem R). Primeiro, esta edição traz dois novos capítulos ao leitor, os quais buscam descrever os dois tipos de vetor S3 mais importantes da linguagem R, isto é, os fatores (`factor`) e os tipos focados em variáveis de tempo (`Date`, `POSIXct`, `POSIXlt` e `difftime`). Tais capítulos são “Capítulo 11 - Introduzindo fatores (`factor`’s) com `forcats`” e “Capítulo 12 - Introduzindo variáveis de tempo com `lubridate`”.

Já a segunda novidade (e na minha visão, a mais importante) desta edição conciste na introdução de exercícios ao final de cada capítulo, além de suas respostas ao final do livro. Exercícios estão presentes em praticamente todo livro técnico, e representam uma etapa muito importante do aprendizado, ao ajudarem o leitor a aplicar e apreender o conhecimento adquirido. Por esses motivos, essa adição representa um passo extremamente importante para este livro.

Mesmo com essas alterações, ainda existem diversas outras adições programadas para as próximas edições. Em outras palavras, este livro ainda está no início de sua vida e possui um longo caminho pela frente. Por exemplo, ele ainda carece de alguns assuntos muito importantes para a linguagem em sua atualidade, como ferramentas para *functional programming* e *meta-programming*, os quais são certamente os próximos passos do livro.

Porque aprender R? Quais são as suas vantagens?

Computadores e linguagens de programação

O R é um ambiente para computação e análise estatística, que possui uma linguagem de programação própria. Para realizar suas atividades no R, você escreve comandos que estão semanticamente de acordo com as regras e padrões dessa linguagem.

Nós como seres humanos, nos comunicamos uns com os outros através da fala, da escrita, da arte, do conhecimento, e de várias outras ferramentas ao nosso dispor, e sempre que estamos utilizando alguma dessas ferramentas, estamos sempre utilizando uma linguagem, ou uma língua específica. Essa língua pode ser algo como o português ou o inglês, mas também pode ser algo como jargões, ou até o estilo de pintura (aquarela, tinta a óleo, etc.) que confere diferentes pesos e gera diferentes sensações nos observadores de sua obra de arte.

Apesar dessas várias opções, nós não podemos utilizar diretamente essas ferramentas para nos comunicarmos com os nossos computadores, pois eles entendem apenas uma língua (*bytes*), e essa língua é extremamente difícil para nós seres humanos. Por essa razão, as diversas linguagens de programação existentes são uma ferramenta de comunicação, criadas justamente com o intuito de facilitar essa comunicação entre você (como usuário) e o seu computador.

Este livro busca lhe ensinar os fundamentos da linguagem R, e como você pode utilizá-la para se comunicar com o seu computador. Entretanto, essa linguagem é uma ferramenta de comunicação

não apenas para o seu computador, mas também para as pessoas que trabalham com você, ou que acompanham o seu trabalho. Pois o código que você escreve no R, carrega a sua metodologia e os seus resultados, e portanto, pode ser utilizado para comunicar as suas intenções e as suas conclusões em uma análise.

Com isso, é natural pensarmos no trabalho necessário para a compreensão de uma língua completamente nova. Entretanto, as linguagens de programação mais populares, hoje, para análise de dados (Python e R) são linguagens fáceis de se aprender. Pois essas linguagens fizeram escolhas (ao serem criadas) que reduzem muito o seu trabalho, e agilizam o seu aprendizado. Por exemplo, nessas linguagens, você não precisa se preocupar em especificar como você deseja alocar os seus dados em memória (algo que é comumente chamado por *memory management* em ciência da computação), ambas fazem este trabalho por você. Essas linguagens também são linguagens interpretadas, logo, você não precisa se preocupar em compilar o seu código antes de executá-lo.

Velocidade e capacidade de processamento

Em resumo, linguagens como Python e R possuem um nível de abstração mínimo, que facilita muito a sua compreensão e o seu trabalho com elas. Por outro lado, devido a essas escolhas, essas linguagens (Python e R) não são particularmente rápidas se comparadas com outras linguagens que lhe obrigam a especificar cada componente de sua análise, como as linguagens C e C++. Pois o computador tem de reservar um tempo para calcular e compilar essas especificações por você.

Porém, essas linguagens ainda assim são muito mais rápidas do que programas como Excel, e lidam muito melhor com grandes quantidades de dados. Por exemplo, se você usa o Excel em seu trabalho, você provavelmente sabe que as suas versões mais recentes são capazes de abrir arquivos com mais de 1 milhão de linhas. Mas se você já tentou, por exemplo, adicionar uma nova coluna a este arquivo, você rapidamente percebeu que o Excel não foi feito para lidar eficientemente com arquivos desta magnitude.

Com linguagens como o R, você possui uma capacidade de processamento maior, e os seus problemas geralmente se limitam a quantidade de memória que você possui em seu computador. Se você possui memória suficiente para alocar uma tabela com mais de 1 milhão de linhas, o seu trabalho com esses dados será muito mais rápido e eficiente no R. E como os componentes de computadores tem ficado cada vez mais baratos, essa vantagem tende a aumentar com o tempo. Hoje, um cartucho de 16GB de RAM (que já é uma quantidade muito boa de memória) é muito mais barato, do que ele era a 10 anos atrás.

Reprodutibilidade: automatizando processos e reduzindo riscos

Vamos a um exemplo prático! Eu comecei a aplicar a linguagem R, quando ainda trabalhava na Diretoria de Estatística e Informações da Fundação João Pinheiro (FJP) como estagiário. A FJP é uma instituição de pesquisa ligada à Secretaria de Estado de Planejamento e Gestão de Minas Gerais, e é responsável pela produção e divulgação das principais estatísticas econômicas e demográficas do estado de Minas Gerais.

Na época, uma de minhas responsabilidades era a produção de mapas temáticos para os informativos mensais de PIB das regiões intermediárias do estado. Eu poderia facilmente gerar esses mapas, utilizando programas especializados como o QGIS. Porém, o QGIS possui uma desvantagem fundamental em relação ao R, especialmente em uma tarefa simples como essa. Onde cada uma das etapas do processo (importando os dados de PIB, importando os *shapefiles*, escolhendo as cores do mapa, escolhendo os títulos e rótulos, criando uma legenda, etc.) não são salvas em algum lugar. Com isso, eu quero destacar que o mapa que eu crio no QGIS, não é reproduzível!

Essas considerações são muito importantes, pois quase sempre eu tinha que reconstruir o mapa. Seja porque o editorial sugeriu o uso de novas cores, ou porque o tamanho da fonte está pequena, ou principalmente, porque erros podem surgir no processo! Se o mapa gerado pelo QGIS possui um erro, seja por falha humana ou do computador, eu tenho que recomençar o trabalho do zero, pois as etapas do processo não foram salvas de alguma forma.

É tendo essas preocupações em mente, que eu possuo um *script* do R, que guarda todos os comandos necessários para produzirmos esses mapas. Dessa maneira, não apenas cada etapa do processo é contida e salva em cada comando do R utilizado, mas eu também posso reproduzir cada uma dessas etapas (ou comandos), com muita facilidade, ao longo de vários pontos diferentes. Isso significa, por exemplo, que eu posso criar um mapa com as mesmas especificações, para cada uma das 13 regiões intermediárias, em questão de segundos, e utilizando apenas 1 comando.

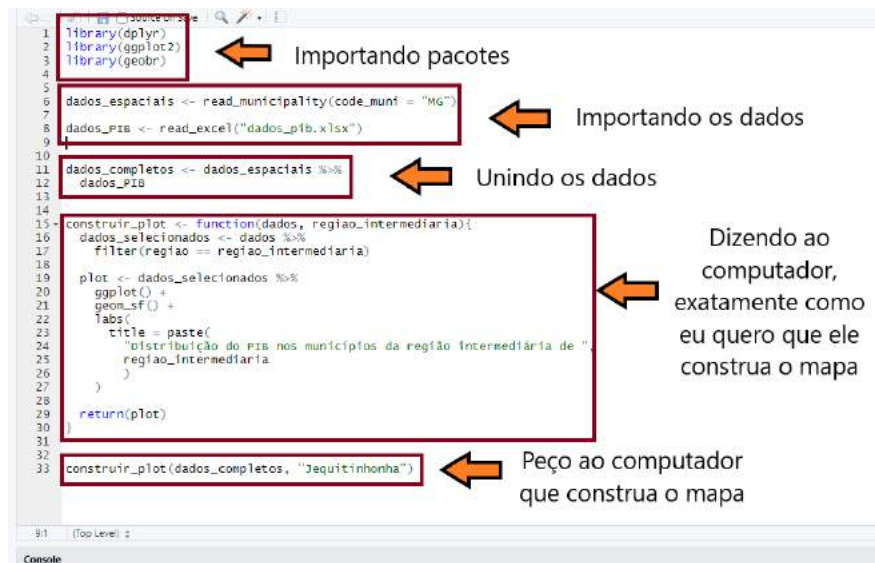
A figura abaixo, é uma representação deste *script*, onde delimito cada uma das etapas que o R realiza para construir esses mapas por mim. Se a nossa equipe descobre um erro no mapa, eu posso voltar ao *script*, e executá-lo parte por parte, e descobrir em qual delas o erro surge. Será que eu errei ao filtrar os dados? Ou o R não conseguiu gerar o gráfico corretamente? Ou será que o erro aparece antes mesmo de eu importar os dados para o R?

A partir do momento em que eu descubro em qual parte de meu *script* o erro ocorre, eu posso corrigir o erro naquele local em específico, e após me assegurar de que tudo está ok, eu posso executar todo o *script* novamente, e assim, o novo mapa contendo as correções aplicadas é gerado em questão de segundos. Dessa forma, eu estou automatizando as etapas repetitivas que possuo em meu trabalho, e não preciso començar do zero caso algum erro ocorra durante o processo.

Neste caso, eu posso inclusive criar alguns processos automatizados que conferem a robustez dos dados, para evitar que erros humanos gerem mais dor de cabeça do que o necessário. Por exemplo, se na minha base de dados, cada linha representa um município de Minas Gerais, eu posso criar um sistema que confere se esta base possui 853 linhas (número total de municípios no estado de Minas Gerais). Como os mapas são geralmente produzidos para cada região intermediária do estado, eu posso também, me certificar que o número de linhas (ou o número de municípios) que compõe cada região intermediária dessa base, estão corretos.

Conexões e API's

A linguagem R possui vários pacotes e interfaces que facilitam a sua conexão com servidores e outras linguagens. Exemplos são os pacotes DBI e odbc, que são muito utilizados para a conexão

Figura 1: Um exemplo de script contendo comandos do R

Fonte: Elaboração própria do autor.

de sua sessão do R, com servidores SQL (*Structured Query Language*). Com essa conexão, você pode puxar resultados de *queries* direto do servidor para a sua sessão do R.

Outro exemplo, é o pacote Rcpp que provê uma boa interface entre o R e a linguagem C++. Com este pacote, você pode misturar comandos em C++ com os seus comandos em R, com o objetivo de utilizar uma linguagem mais rápida (C++) em processos que são, por natureza, muito trabalhosos para o seu computador. Além disso, tanto o Python quanto o R, possuem interfaces para se comunicar um com o outro. Isto é uma ferramenta muito poderosa! Pois você pode se aproveitar do melhor que as duas principais linguagens utilizadas em análise de dados podem oferecer. O R possui um arsenal estatístico melhor do que o Python, porém, ele não possui a conectividade e amplitude de aplicações que o Python oferece. Logo, no caso do R, você pode utilizar o pacote *reticulate*, que fornece uma boa interface para o interpretador do Python.

Para mais, grande desenvolvimento tem sido empregado em serviços web. Uma área que até pouco tempo, possuía pouco suporte dentro da linguagem R. Hoje, você já pode criar sites (pacote *blog-down*) e dashboards interativos (pacote *shiny*) com os recursos disponíveis. Também há pacotes como *httr* e *rvest*, que possibilitam a realização de atividades de *web scrapping*. Além dos pacotes *xml2* e *jsonlite*, que permitem a leitura de dados em XML e JSON, respectivamente. Para esse tópico, você pode descobrir mais pacotes na seção de [Web Technologies do CRAN R](#).

Por último, o time da Microsoft, também tem desenvolvido interfaces em seus serviços da Azure Cloud Computing, permitindo que você utilize R em seus projetos na plataforma. Caso esteja interessado nisso, você pode consultar a página da empresa sobre este serviço².

²<https://docs.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/r-developers-guide>

Comunidade

O R é uma linguagem gratuita e *open source* e, por isso, o seu crescimento como linguagem depende não apenas da fundação que a mantém e a atualiza (*R Foundation*), mas também depende de sua comunidade que está o tempo todo discutindo, inovando e abrindo novos caminhos, tudo isso de forma aberta e gratuita. Esta obra é uma contribuição a essa comunidade e um convite a você. Venha para a comunidade de R!

Portanto, a comunidade é um dos principais ativos da linguagem R (e também do Python). Grande parte dessa comunidade, está concentrada no [Twitter](#). Mas essa comunidade também está muito presente em blogs, comentando novas soluções e recursos para a linguagem (sendo o [Tidyverse blog](#), [Rweekly](#) e [ROpenSci](#) os principais exemplos) e, com isso, você pode se manter atualizado sobre o que a linguagem oferece. Por outro lado, parte desses blogs, possuem um foco maior em tutoriais, e representam assim, um local em que você sempre pode aprender mais sobre o R (o principal exemplo dessa categoria se trata do [R-Bloggers](#)).

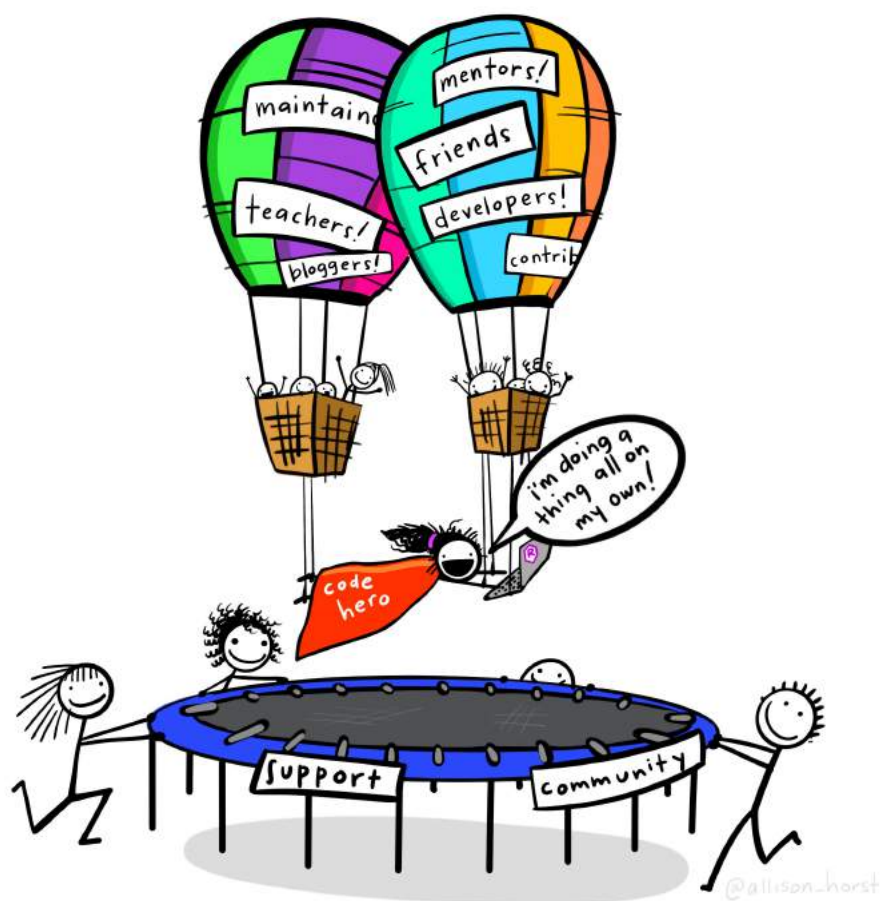
Recentemente, um novo e excelente centro de discussão foi criado pela comunidade, denominado [R4DS Online Learning Community](#), um nome que claramente se refere a obra de [Wickham e Grolemund \(2017\)](#). Esse é um ótimo local para criar conversas com membros da comunidade, e pedir por ajuda em algum problema que você esteja enfrentando.

Além disso, a comunidade de R também possui forte presença no [StackOverflow](#), que é comumente caracterizado como o principal canal de dúvidas e de ajuda em diversas linguagens de programação. Logo, se você não sabe como realizar um processo, ou não consegue descobrir de onde um erro está surgindo em seu *script*, você pode pedir por ajuda da comunidade ao postar uma pergunta, ou encontrar uma pergunta parecida com o seu problema que já foi respondida no [StackOverflow](#).

No caso do Brasil, a principal força motriz de nossa comunidade provavelmente se encontra nos capítulos brasileiros do [R-Ladies global](#), além do blog [Curso-R](#). Por exemplo, temos os encontros mensais online realizados pelo [capítulo de São Paulo](#), além dos bons tutoriais escritos pelo [capítulo de Belo Horizonte](#). Existem também, outros capítulos no Brasil³, que também realizam alguns encontros.

³Você pode consultar a lista completa dos capítulos brasileiros na página principal do [R-Ladies global](#).

Figura 2: Code Hero por Allison Horst



Fonte: Allison Horst GitHub.

Capítulo 1

Noções Básicas do R

1.1 Uma descrição do R

1.1.1 História do R

A linguagem R, nasceu durante a década de 90, inicialmente como um projeto de pesquisa de Ross Ihaka e Robert Gentleman, ambos estatísticos e pesquisadores associados na época ao departamento de estatística da Universidade de Auckland ([IHAKA; GENTLEMAN, 1996](#)). Porém, as origens da linguagem R retornam a década de 70, com o desenvolvimento da linguagem S, em um dos mais importantes laboratórios de pesquisa do mundo, a Bell Labs ([PENG, 2015](#)).

Pois como foi descrito por [Ihaka e Gentleman \(1996\)](#), a linguagem R foi desenvolvida com fortes influências das linguagens S e Scheme. Sendo que a própria sintaxe da linguagem R, se assemelha muito a da linguagem S. Por isso, muitos autores como [Peng \(2015\)](#) e [Chambers \(2008\)](#), caracterizam a linguagem R como um dialeto da linguagem S. Segundo [Ihaka e Gentleman \(1996\)](#) a linguagem S representava uma forma concisa de se expressar idéias e operações estatísticas para um computador e, por isso, foi uma fonte de inspiração importante para o R. Em outras palavras, comparado às demais linguagens, a linguagem S oferecia uma sintaxe mais atrativa e confortável para estatísticos executarem as suas ideias, e grande parte dessa sintaxe, foi transportada para o R.

1.1.2 R

R é um *software* estatístico que oferece um ambiente para análise interativa de dados, e que conta com uma poderosa linguagem de programação, e é dessa linguagem que vamos tratar neste livro. Diferente de outras linguagens como C e C++, que são linguagens compiladas, a linguagem R é uma linguagem interpretada. Isso significa, que para trabalharmos no R, vamos estar constantemente enviando comandos escritos para o Console do programa, e esse Console vai avaliar os comandos que enviarmos (segundo as “regras gramaticais” da linguagem R), antes de executá-los.

Logo, o console é o coração do R, e a mais importante ferramenta do programa ([ADLER, 2010, p.11](#)), pois é nele que se encontra o interpretador que vai avaliar e executar todos os nossos coman-

dos. O uso de uma linguagem de programação, representa uma maneira extremamente eficiente de se analisar dados, e que de certa forma, adquire um aspecto interativo no R, ou cria uma sensação de que estamos construindo (interativamente) uma conversa com o console. Ou seja, o trabalho no R funciona da seguinte maneira: 1) você envia um comando para o console; 2) o comando é avaliado pelo console e é executado; 3) o resultado desse comando é retornado pelo console; 4) ao olhar para o resultado, você analisa se ele satisfaz os seus desejos; 5) caso não, você faz ajustes em seu comando (ou utiliza um comando completamente diferente), e o envia novamente para o console; e assim, todo o ciclo recomeça.

1.1.3 O sistema e universo do R

O universo do R pode ser dividido em duas partes, sendo elas:

1. O sistema “básico” do R, que é composto pelos pacotes básicos da linguagem. Esses pacotes são a base da linguagem R, e são comumente chamados pela comunidade, por base R. Pois diversas das funções básicas do R, advêm de um pacote chamado base. Lembrando que você pode baixar e instalar esses pacotes básicos, pelo site do [Comprehensive R Archive Network \(CRAN R\)](#).
2. Todo o resto, ou mais especificamente, todos os pacotes externos ao sistema “básico”, desenvolvidos pelo público em geral da linguagem. A grande maioria desses pacotes também estão disponíveis através do [Comprehensive R Archive Network \(CRAN R\)](#), mas alguns outros estão presentes apenas em outras plataformas, como o [GitHub](#).

Todas as funcionalidades e operações disponíveis no R, são executadas através de suas funções, e essas funções são divididas em “pacotes”. O sistema “básico” do R, contém um conjunto de pacotes que oferecem as funcionalidades básicas da linguagem. Alguns desses pacotes básicos, são base (fornece funções de uso geral) e stats (fornece funções para análises e operações estatísticas). Caso você precise de funcionalidades que vão além do que está disponível neste sistema “básico”, ou neste conjunto de pacotes “básicos” do R, é neste momento em que você precisa instalar outros pacotes que estão fora desse sistema “básico”, e que oferecem funções que possam executar as funcionalidades que você deseja. Vamos dissecar alguns desses pacotes “externos” ao longo deste material, com especial atenção ao conjunto de pacotes fornecidos pelo [tidyverse](#).

Pelo fato do R ser gratuito e *open source*, várias pessoas estão constantemente desenvolvendo novas funcionalidades, e efetivamente expandindo o universo da linguagem R. Esses pacotes desenvolvidos pelos próprios usuários da linguagem, servem como grande apoio ao trabalho de outros usuários. Ou seja, se você possui um problema a sua frente, é muito provável que alguém tenha enfrentado o mesmo problema, ou algo próximo, e que tenha desenvolvido uma solução para aquele problema no formato de um pacote do R. Assim, você pode resolver os seus problemas, com a ajuda do trabalho de outras pessoas que passaram pelas mesmas dificuldades.

1.1.4 RStudio

O RStudio, é um Ambiente de Desenvolvimento Integrado (*Integrated Development Environment* - IDE, em inglês) para o R. Em síntese, esse programa oferece um ambiente com diversas melhorias, atalhos e ferramentas que facilitam de maneira expressiva, o seu trabalho com o R. Algumas dessas funcionalidades incluem: indentação automática, realces de código, menus rápidos para importação e exportação de arquivos, além de diversos atalhos de teclado úteis. Sendo portanto, uma ferramenta muito recomendada para qualquer usuário que venha a trabalhar com a linguagem R (GILLESPIE; LOVELACE, 2017).

Para encontrar mais detalhes sobre o programa, você pode consultar o [site oficial do RStudio](#).

1.2 Introdução ao R e RStudio: noções básicas

1.2.1 Executando comandos: Console

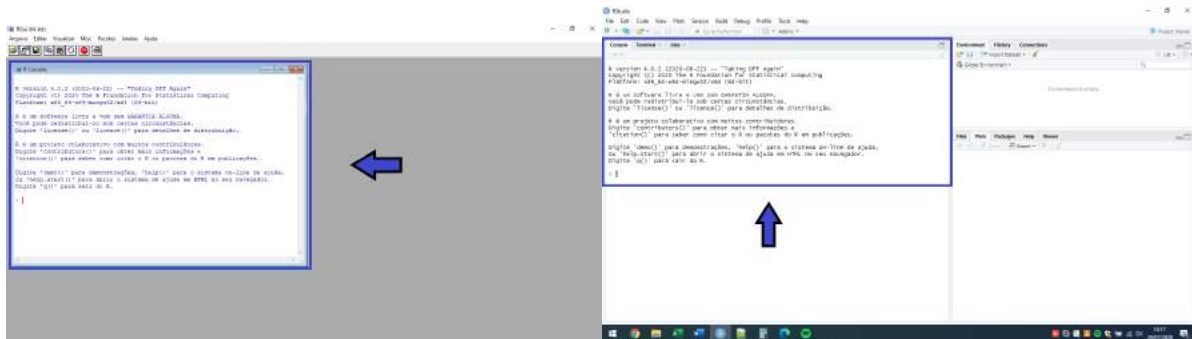
Você trabalha no R através de sua linguagem de programação. Você importa os seus dados, remove ou acrescenta colunas, reordena a sua base, constrói gráficos, e estima os seus parâmetros, através de comandos escritos que devem ser interpretados e executados pelo Console. Qual o porquê de tudo isso? Por que precisamos de um console para interpretar os nossos comandos? A resposta se encontra no fato de que o seu computador não fala a sua língua!

Ou seja, o seu computador não sabe o que os verbos “ordenar” e “selecionar” significam, estejam eles em qualquer língua humana que você conseguir imaginar agora. Pois o seu computador, só fala e compreende uma única língua, que é extremamente difícil para nós seres humanos, que são os *bits* ou *bytes* de informação. Se quisermos nos comunicar com o nosso computador, e passarmos instruções e comandos para serem executados por ele, nós devemos repassar essas informações como *bits* de informação. Com isso, o trabalho do Console, e principalmente do interpretador presente nele, é o de traduzir os seus comandos escritos na linguagem R (que nós seres humanos conseguimos entender), para comandos em *bits*, de forma que o seu computador possa compreender o que você está pedindo a ele que faça.

Tanto no programa padrão do R, quanto no RStudio, o console se localiza a esquerda de sua tela, como mostrada na figura 1.1:

Ao olhar para o Console, você pode perceber que em sua parte inferior, nós temos no início da linha um símbolo de “maior que” (>). Esse símbolo, significa que o Console está pronto e esperando por novos comandos a serem interpretados. Ou seja, você coloca os seus comandos à frente deste símbolo, e em seguida, você aperta Enter para confirmar o envio dos comandos. Assim, os comandos serão avaliados, e o console vai lhe retornar o resultado destes comandos. Há algumas ocasiões em que o console vai apenas executar os comandos, e não irá lhe mostrar automaticamente o resultado. Isso geralmente ocorre quando você está salvando os resultados desses comandos em um objeto (iremos aprender sobre eles mais a frente).

Como um exemplo clássico, eu posso utilizar o R como uma simples calculadora, ao escrever o

Figura 1.1: Consoles no R padrão (à esquerda) e no RStudio (à direita)

Fonte: Elaboração própria do autor.

comando “1 + 3” no Console (e apertar a tecla Enter), e como não estou salvando o resultado dessa soma em algum objeto, o console me mostra automaticamente o resultado dessa operação.

```
1 + 3
```

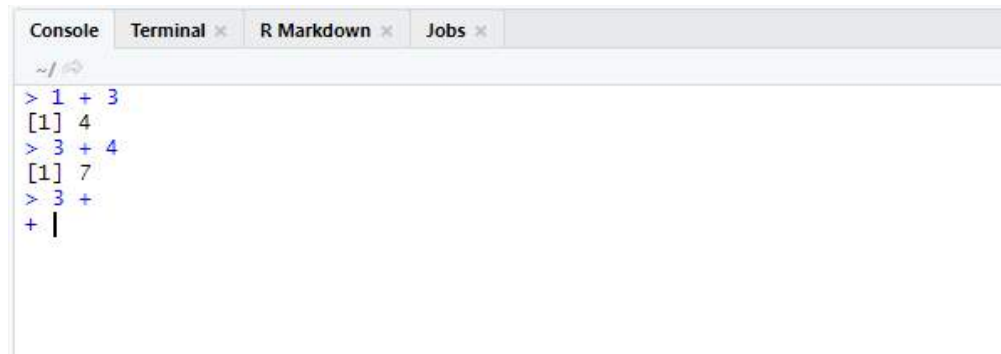
```
## [1] 4
```

Vale destacar, que todo comando que você escrever no Console, deve estar completo para ser avaliado. Dito de outra forma, quando você escreve no Console, algum comando que ainda está incompleto de alguma forma (por exemplo, que ainda está faltando fechar algum par de parênteses, ou está faltando uma vírgula, ou está faltando algum valor a ser fornecido), e você aperta Enter para ele ser avaliado, o símbolo > do Console, será substituído por um +, te indicando que ainda falta algo em sua expressão. Neste caso, o Console ficará esperando até que você escreva o restante, e complete o comando, como mostrado na figura 1.2. Em uma situação como essa, você pode abortar a operação, e reescrever do início o seu comando, ao apertar a tecla Esc de seu computador.

1.2.2 Comentários

O R possui diversos caracteres especiais, e que sofrem ou geram efeitos distintos ao serem avaliados. Um desses caracteres, é a *hash* (#), que no R, representa o início de um comentário. Ou seja, todo e qualquer comando, letra ou expressão escrita após o símbolo # (incluindo o próprio símbolo #), será completamente ignorado pelo Console. Portanto, o símbolo # constitui uma forma útil de incluirmos anotações e comentários em nossos comandos. Por exemplo, você talvez tenha dificuldade de lembrar o que cada função faz, e por isso, você pode utilizar o símbolo # para inserir pequenas descrições e lembretes ao longo de seus comandos, para relembrá-lo o que cada função faz.

```
# A função sum() serve para somar um
# conjunto de números.
sum(1,2,3,4,5)
```

Figura 1.2: Expressões incompletas

```
Console Terminal x R Markdown x Jobs x
> 1 + 3
[1] 4
> 3 + 4
[1] 7
> 3 +
+ |
```

Fonte: Elaboração própria do autor.

[1] 15

1.2.3 Comandos e resultados

O símbolo de “maior que” (>) no Console, também representa uma forma útil de você diferenciar o que é um comando a ser interpretado pelo R, e o que foi retornado pelo R como o resultado desse comando. Ou seja, todo bloco de texto em seu Console, que estiver logo à direita do símbolo >, representa um bloco de comandos a serem avaliados (ou que já foram avaliados) pelo R. Em contrapartida, todo texto que não possuir o símbolo > à sua esquerda, representa o resultado do comando anterior, ou então, uma mensagem de erro referente a esse comando anterior.

Uma outra forma útil de identificar os resultados de seus comandos, é perceber que eles sempre vem acompanhados por algum índice numérico no início de cada linha. Esse índice pode estar dentro de um par de colchetes (como [1]), ou pode estar livre, como no resultado da função `data.frame()` apresentado na figura 1.3. Perceba que esses números são apenas índices, logo, eles não fazem parte do resultado de seus comandos, e são apenas valores que marcam o início cada linha de seu resultado.

1.2.4 Histórico de comandos

O Console possui uma memória dos comandos que você executou anteriormente. Tanto que esses comandos e seus resultados, permanecem visíveis ao navegarmos pelo Console. Porém, você também pode navegar pelos comandos previamente executados, ao utilizar a seta para cima (↑) de seu teclado, quando estiver no Console. Através dessa tecla, os comandos executados anteriormente são apresentados na linha de inserção de códigos do próprio Console.

Porém, você também pode visualizar de forma mais eficiente o seu histórico de comandos, ao acessar a janela History do RStudio, que fica na parte direita e superior de sua tela, como mostrado na figura 1.4. Uma outra forma de abriremos essa janela, está na função `history()`. Com essa função,

Figura 1.3: Comandos e seus respectivos resultados no Console

```

~/Projeto curso R/Curso-R/
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> "Olá Mundo"
[1] "Olá Mundo"
> data.frame(id = LETTERS[1:8], x = rnorm(8), y = rnorm(8))
  id      x      y
1 A -0.9121533 0.4033139
2 B  2.5811986 1.5944208
3 C -1.2480110 -1.8605733
4 D -1.0861465 -1.2915975
5 E -0.6571215  2.0411512
6 F  0.6769102  1.0281839
7 G  1.6168743  0.5322960
8 H -1.2404146 -1.8268386
> mean(rnorm(40, 50, 12))
[1] 51.59429
>

```

Comandos
Resultados

Fonte: Elaboração própria do autor.

you can determine how many previous commands should be displayed in this window.

```

# Exibir os últimos 10 comandos executados
history(10)

```

Figura 1.4: Aba History - Quadrante superior direito

```

Environment History Connections Git Tutorial
.m2edia <- Z
.m2edia
history(10)
20%%8
20/8
20 %% 8
517 %% 100
3*5
15/5
20 %/% 8
20 %% 8

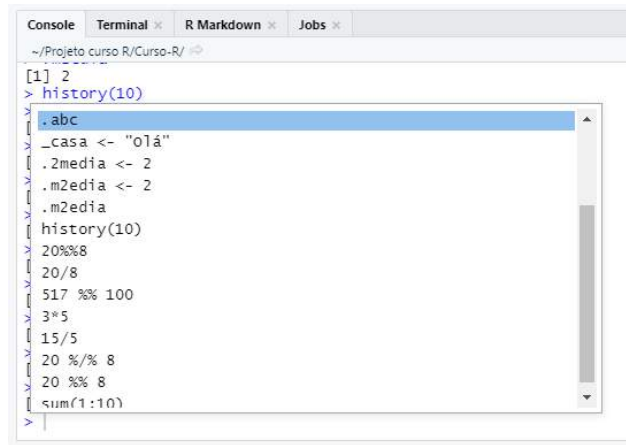
```

Fonte: Elaboração própria do autor.

For more, you can also visualize this command history, by means of a small window open in your Console, as in figure 1.5. When you are in the console, you can access this window, by pressing the keys `Ctrl + ↑`.

1.2.5 Operações matemáticas básicas

R can be used as a simple calculator, through its arithmetic operators.

Figura 1.5: Histórico de comandos - Console

Fonte: Elaboração própria do autor.

Simples Adição

```
3 + 15
```

```
## [1] 18
```

Multiplicação

```
3 * 125
```

```
## [1] 375
```

Potenciação

```
3 ^ 4
```

```
## [1] 81
```

Miscelânea de operadores

```
((4.505 * 100)/ 5) + 0.015
```

```
## [1] 90.115
```

Você irá rapidamente perceber que esses operadores são extremamente úteis e estão por toda parte, sendo utilizados em diversas outras operações muito mais complexas. Por isso, é importante que você leve um tempo se familiarizando com esses operadores. Temos na tabela 1.1, uma lista dos principais operadores aritméticos, além de alguns comandos no R, que exemplificam o seu uso.

Tabela 1.1: Operadores aritméticos do R

Operação	Operador no R	Código exemplo	Resultado
Adição	+	5 + 5	10
Subtração	-	15 - 5	10
Divisão	/	15 / 5	3
Multiplicação	*	3 * 5	15
Exponenciação	^ ou **	2 ^ 5	32
Parte inteira da divisão	%/%	20 %/% 8	2
Resto da divisão	%%	20 %% 8	4

Fonte: Elaboração própria do autor.

1.3 Introdução a objetos

Uma das principais características do R, é que ele é uma linguagem orientada a objetos (*object oriented*). Objetos são o método que o R possui para guardar os valores, funções e resultados que você produz. Como foi posto por [Adler \(2010, p.50\)](#), todo código do R, busca utilizar, manipular ou modificar de alguma forma, um objeto do R. Logo, quando você estiver trabalhando com seus dados no R, você estará constantemente aplicando operações e transformações sobre os objetos onde seus dados estão guardados, de uma forma interativa e dinâmica.

Para que um objeto seja criado, o R necessita de uma forma de referenciar aquele objeto, ou em outras palavras, uma forma de reconhecer o objeto ao qual você está requisitando. Esse mecanismo conciste fundamentalmente de um nome ([CHAMBERS, 2008, p.24](#)). Ou seja, todo objeto no R, possui um nome, e será através desse nome, que você será capaz de acessar esse objeto. Portanto, para você salvar todo e qualquer resultado ou valor no R, você precisa obrigatoriamente salvá-lo dentro de um objeto, isto é, dar um nome a esse resultado ou valor que você está gerando.

No exemplo abaixo, eu estou guardando a minha idade em um objeto chamado `idade_pedro`. Dessa forma, quando eu precisar deste número em algum momento de minha análise, eu preciso apenas chamar pelo nome onde guardei este número, ou nos termos do R, pelo nome dei ao objeto onde guardei este número.

```
idade_pedro <- 22
```

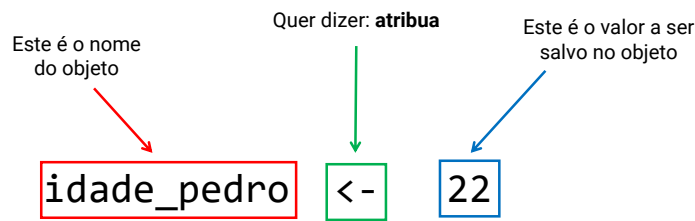
Após criarmos o objeto de nome `idade_pedro`, eu posso acessar o valor que foi salvo nele, ao chamar pelo nome do objeto no Console.

```
idade_pedro
```

```
## [1] 22
```

Sempre que você estiver criando um objeto, ele irá seguir essa estrutura acima. Você possui primeiro o nome do objeto, depois o símbolo de assignment (<-), e por último, o valor (ou o conjunto de valores) que você quer guardar dentro deste objeto. Independente do quê, ou, do porquê, o código à direita do símbolo de assignment faz, ao ver essa estrutura, você sabe de antemão que esses comandos estão criando um objeto.

Figura 1.6: Estrutura necessária para criar um objeto



Em português, você está pedindo ao R:
Atribua ao objeto de nome idade_pedro, o valor de 22

Fonte: Elaboração própria do autor.

Nós podemos sobrepor o valor guardado em um objeto, ao atribuir um novo valor a este objeto. Neste caso, estaríamos perdendo o valor que salvamos anteriormente neste objeto. Como exemplo, se eu atribuir o texto “Importado” ao objeto idade_pedro. Após este novo comando, se chamarmos pelo nome do objeto, o R irá lhe mostrar o novo texto que acabamos de guardar, e o número 22 que estava anteriormente guardado nele, se perdeu.

```
idade_pedro <- "Importado"
```

```
idade_pedro
```

```
## [1] "Importado"
```

Caso você tenha que sobrepor o valor de um objeto, mas você não quer perder o valor que está salvo em nele, você deve conectar este valor a um novo objeto. Se um valor não está conectado a um nome, o R vai jogar este valor fora, por isso, precisamos de uma nova conexão até ele, ou em outras palavras, precisamos conectá-lo a um novo nome. Dessa forma, podemos tranquilamente sobrepor o valor guardado em idade_pedro, pois agora, o valor 22 está guardado em um outro objeto.

```
idade_pedro <- 22
```

```
numero_importante <- idade_pedro

idade_pedro <- "Importado"

# Ao chamar pelo nome de
# ambos os objetos, temos dois valores
# diferentes

idade_pedro

## [1] "Importado"

numero_importante

## [1] 22
```

1.3.1 Como nomear um objeto

Como foi destacado por [Wickham e Grolemund \(2017\)](#), existem regras sobre como você pode nomear os seus objetos no R. Segundo [R Core Team \(2020a, p.4\)](#), o nome de um objeto, pode conter qualquer símbolo alfanumérico (qualquer letra ou número), inclusive letras acentuadas. Sendo que o nome desse objeto, deve obrigatoriamente se iniciar por uma letra, ou por um ponto (.), como por exemplo, os nomes: População; dados.2007; .abc; media_1990. Porém, um nome não pode começar por um número, logo, um nome como 1995populacao, não é permitido. Também não é possível, que se inicie um nome por um ponto (.) caso ele seja seguido por um número. Logo, você não pode criar um objeto com o nome .2media, mas você pode criar um objeto que possua o nome .m2edia ou .media2.

Em suma, o nome de um objeto pode conter os seguintes tipos de caractere:

- Letras.
- Números.
- _ (*underline*).
- . (ponto).

Além disso, o nome de um objeto pode se iniciar com um:

- Letra.
- . (ponto, desde que não seja seguido por um número).

Porém, o nome de qualquer objeto, **não deve começar** por um:

- _ (*underline*).
- Número.
- . (ponto) seguido de um número.

Pode ser difícil pensar em um nome para os seus objetos. Mas a melhor alternativa, é sempre dar um nome claro e descritivo aos seus objetos, mesmo que esse nome possa ficar muito extenso. Por exemplo, `microdados_pnad_2020` para uma base de dados contendo os microdados da PNAD de 2020; ou `vetor_idade`, para um vetor que contém as idades das pessoas que foram entrevistadas em uma pesquisa.

1.3.2 O R é *case-sensitive*

O R é uma linguagem *case-sensitive*. Isso significa, que ele é capaz de diferenciar a capitalização de sua escrita. Logo, um objeto chamado `a`, é um objeto completamente diferente de um objeto chamado `A`. Veja o exemplo abaixo.

```
casa <- 10 ^ 2  
cAsa <- 2 + 2
```

```
casa
```

```
## [1] 100
```

```
cAsa
```

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

Como visto, os objetos `casa` e `cAsa` contêm valores diferentes, e portanto, representam objetos distintos.

1.3.3 Como utilizar objetos

Um objeto é de certa forma, uma referência até um certo conjunto de valores, e você utiliza, ou acessa essa referência, através do nome que você deu a esse objeto. Logo, sempre que você quiser utilizar os valores que estão guardados em algum objeto (seja dentro de alguma função ou em alguma operação específica), você precisa apenas utilizar o nome que você deu a esse objeto.

Por exemplo, se eu quero somar um conjunto de valores guardados em um objeto chamado `vec_num`, eu posso fornecer o nome deste objeto à função `sum()`.

```
vec_num <- c(2.5, 5.8, 10.1, 25.2, 4.4)
```

```
soma <- sum(vec_num)
```

```
soma
```

```
## [1] 48
```

1.4 Funções (noções básicas)

Como destacado por [Chambers \(2016\)](#), até as funções que você utiliza, são objetos no R. A grande maioria das funções são escritas e utilizadas, segundo o formato abaixo. Portanto, sempre que você for utilizar uma função no R, você deve escrever o nome dessa função, e em seguida, abrir um par de parênteses. Dentro destes parênteses, você irá fornecer os argumentos (ou *input's*) que serão utilizados pela função para gerar o seu resultado.

```
nome_da_função(lista_de_argumentos)
```

Os operadores aritméticos utilizados até aqui (+, -, *, etc.) também são funções para o R, porém, eles representam um tipo especial de *função*. Pois nós podemos posicionar os seus argumentos, ao redor desses operadores (Ex: 2 + 3). Por outro lado, nós podemos escrever esses operadores (ou essas funções), da forma “tradicional”, ou como as demais funções no R são escritas, o que é demonstrado logo abaixo. Perceba que pelo fato do nome da função (a função que representa o operador +), se iniciar por um símbolo que não respeita as regras que definimos anteriormente (sobre como nomear um objeto), para nos referirmos a esse objeto, ou a essa função, nós devemos contornar o nome dessa função por acentos graves.

```
# 0 mesmo que 2 + 3  
'+'(2, 3)
```

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

```
# 0 mesmo que 12 + 8  
'+'(12, 8)
```

```
## [1] 20
```

Os argumentos da função identificam os dados que serão transformados, ou representam especificações que vão modificar o comportamento da função, ou modificar a metodologia de cálculo utilizada. Dessa forma, nós definimos os argumentos das funções (i.e., incluímos as especificações desejadas) para que possamos obter os resultados de acordo com as nossas necessidades. Sendo que a *lista_de_argumentos*, corresponde a uma lista onde cada argumento é separado por uma vírgula (,), como no exemplo abaixo.

```
exemplo_função(argumento1 = valor_argumento1, argumento2 = valor_argumento2)
```

Um argumento pode ser um símbolo, que contém um valor específico (ex.: `argumento1 = valor_argumento1`) ou o argumento especial `'...'`, que pode conter qualquer número de argumentos (geralmente, o argumento especial é encontrado em funções em que a quantidade de argumentos que será passada é desconhecida). Algumas funções possuem valores padrões em seus argumentos.

Em outras palavras, caso você não defina algum valor específico para este tipo de argumento, a função vai utilizar um valor pré-definido para esse argumento. Usualmente, os valores padrão são os valores mais comuns, que utilizam as metodologias mais conservadoras ou tradicionais de cálculo da função.

Por exemplo, a função `sum()` possui o argumento `na.rm`, que define se os valores NA presentes em um objeto, devem ser ignorados ou não durante o cálculo da soma. Por padrão, esse argumento é configurado para FALSE (falso). Isso significa, que qualquer valor NA que estiver presente no objeto a ser somado, vai alterar o comportamento da soma executada por `sum()`. Por isso, se quisermos ignorar os valores NA durante o cálculo da soma, nós precisamos definir explicitamente o argumento `na.rm` para TRUE (verdadeiro).

```
vec <- c(1.2, 2.5, 3, NA_real_, 7.6)
```

```
sum(vec)
```

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

```
sum(vec, na.rm = TRUE)
```

```
## [1] 14.3
```

Ao definirmos os valores a serem utilizados em cada argumento de uma função, nós não precisamos determinar o nome do argumento a ser utilizado. Como exemplo, veja a função `rnorm()` abaixo. O primeiro argumento (`n`) da função, define o número de observações a serem geradas; o segundo (`mean`), define a média desses valores; e o terceiro (`sd`), define o desvio padrão que esses valores vão seguir ao serem gerados.

```
# A função rnorm() e seus argumentos
rnorm(n, mean, sd)
```

Quando nós não definimos explicitamente o nome do argumento que estamos utilizando, o R vai conectar o valor que fornecemos, de acordo com a ordem que os argumentos aparecem na função. Ou seja, o primeiro valor, será conectado ao primeiro argumento da função. Já o segundo valor, será conectado ao segundo argumento da função. E assim por diante. Isso significa, que se quisermos configurar algum argumento, fora da ordem em que ele aparece na função, nós teremos que explicitar o nome do argumento a ser utilizado.

```
rnorm(10, 15, 2.5)
```

```
## [1] 8.394339 13.901849 17.968768 15.609148 13.661457 15.400751 15.093411
```

```
## [8] 15.225535 17.059474 14.750358
```

```
rnorm(n = 10, sd = 2.5, mean = 15)
```

```
## [1] 17.49823 13.57708 14.33244 14.98843 13.41891 15.69303 14.31771
```

```
## [8] 14.27685 12.94187 14.52364
```

1.5 Erros e ajuda: como e onde obter

Ao começar a aplicar o conhecimento exposto neste livro, você rapidamente irá enfrentar situações adversas, onde vão surgir muitas perguntas das quais eu não ofereço uma resposta aqui. Por isso, é muito importante que você conheça o máximo de recursos possíveis, dos quais você pode consultar e pedir por ajuda (WICKHAM; GROLEMUND, 2017).

Hoje, a comunidade internacional de R, é muito grande, e há diversos locais onde você pode encontrar ajuda, e aprender cada vez mais sobre a linguagem. Nessa seção, vamos explicar como utilizar os guias internos do R e do RStudio, além de algumas técnicas de pesquisa e de perguntas que podem te ajudar a responder as suas dúvidas.

1.5.1 Ajuda Interna do R: help e ?

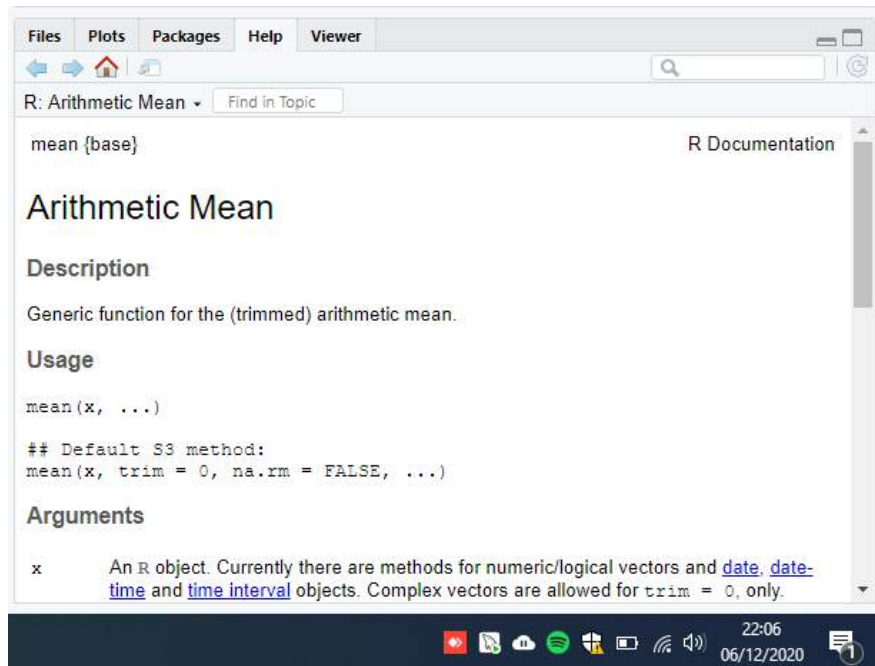
Toda função no R, possui uma documentação interna, que contém uma descrição completa (ou quase sempre completa) da função. Essas documentações são muitas vezes úteis, especialmente para descobrirmos os argumentos de uma função, ou para compreendermos que tipo de valores devemos utilizar em um certo argumento, ou então, em ocasiões mais específicas, para adquirirmos um conhecimento mais completo sobre o comportamento de uma função. Para acessar essa documentação, você pode anteceder o nome da função com o operador `?`, ou então, utilizar a função `help()` sobre o nome da função de interesse. Como exemplo, com os comandos abaixo, você pode consultar a documentação interna da função `mean()`.

```
# Usando `help()`  
help("mean")  
# Usando `?`  
?mean
```

Se você estiver no programa padrão do R, ao executar um desses comandos, um arquivo HTML contendo a documentação será aberto em seu navegador. Mas se você estiver no RStudio, a documentação será aberta na janela de Help do próprio RStudio, localizada no quadrante direito e inferior de sua tela, como mostrado na figura 1.7.

A documentação interna de uma função, lhe dá uma descrição completa sobre quais tipos de valores devem (ou podem) ser inseridos em cada argumento da função. Entretanto, caso você esteja apenas em dúvida sobre os nomes dos argumentos de uma função, você pode rapidamente sanar essa dúvida, ao utilizar a função `args()` sobre o nome da função. Com essa função, uma estrutura é retornada, contendo a palavra chave `function`, e a lista de argumentos da função dentro de um par de parênteses. Porém, se a função de interesse possui diferentes métodos (como é o caso da função `mean()`, e de muitas outras funções), é muito provável que o resultado da função `args()` será de pouca utilidade, a menos que você pesquise por um método específico da função.

```
args("mean")
```


Figura 1.7: Documentação interna da função de média no RStudio

Fonte: Elaboração própria do autor.

```
## function (x, ...)
## NULL
```

Veja no exemplo abaixo, que ao selecionarmos o “método padrão” da função de média (`mean.default()`), dois novos argumentos foram retornados (`trim` e `na.rm`) pela função `args()`.

```
args("mean.default")

## function (x, trim = 0, na.rm = FALSE, ...)
## NULL
```

Por esses motivos, a documentação interna representa uma fonte mais completa e segura de consulta sobre uma função. Como exemplo, na figura 1.8 podemos ver a seção de Arguments, da documentação interna da função `mean()`. Nessa seção, podemos encontrar uma descrição sobre o que cada argumento faz e, principalmente, sobre que tipo de valor cada um desses argumentos é capaz de receber. Vemos abaixo, pela descrição do argumento `x`, que a função `mean()` possui métodos específicos para vetores numéricos, lógicos, de data, de data-hora e de intervalos de tempo. Com essas informações, nós sabemos, por exemplo, que a função não possui métodos para vetores de texto. Também podemos deduzir dessa descrição, que a função `mean()` é capaz de lidar apenas com vetores, e portanto, o uso de `data.frame`'s e listas está fora de cogitação.

Figura 1.8: Seção de argumentos da documentação interna da função de média

Arguments

x An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.

... further arguments passed to or from other methods.

Fonte: Elaboração própria do autor.

1.5.2 Um exemplo clássico de ajuda interna

Um exemplo clássico em que a ajuda interna do R é bem útil, se encontra na função `round()`, que utilizamos para arredondar valores numéricos de acordo com um número de casas decimais.

```
# Arredondar 3.1455 para duas casas decimais
round(3.1455, digits = 2)

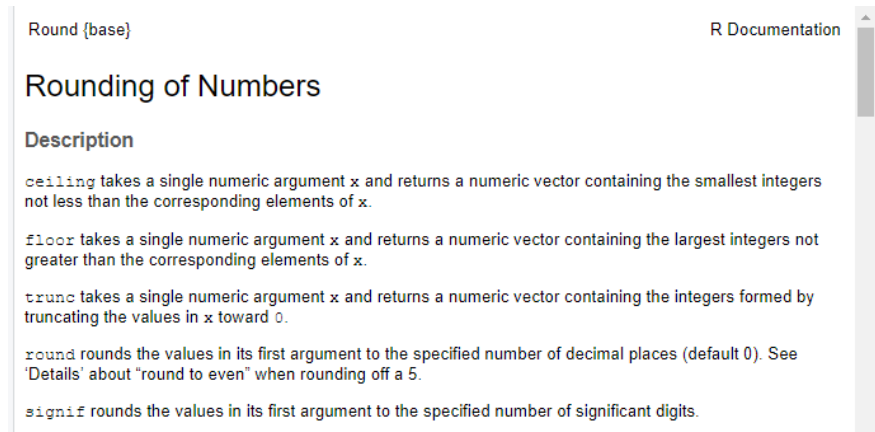
## [1] 3.15
```

Porém, há diversas maneiras de se arredondar um número, e você talvez se pergunte quais desses métodos estão disponíveis no R. Para responder a essa pergunta, você talvez pense em procurar por mais detalhes sobre a função `round()` em sua documentação interna. Temos o início dessa documentação na figura 1.9, e a primeira coisa que chama atenção é a lista de funções irmãs de `round()`. Ou seja, possuímos nessa lista, 5 funções diferentes (`ceiling()`, `floor()`, `trunc()`, `round()` e `signif()`), que buscam aplicar diferentes métodos de arredondamento.

Ao olharmos para a descrição da função `floor()` (“*takes the largest integer not greater than the corresponding value of x* ”, ou “seleciona o maior número inteiro que não é maior do que o valor correspondente em x ”), podemos compreender que essa função busca sempre arredondar um número para baixo, independentemente de qual número esteja presente na última casa decimal. Também podemos entender pela descrição da função `ceiling()`, que ela executa justamente o processo contrário (“*takes the smallest integer not less than the corresponding value of x* ”, ou “seleciona o menor número inteiro que não é menor que o valor correspondente de x ”), e arredonda qualquer número sempre para cima.

```
vec <- c(0.4, 2.5, 3.7, 3.2, 1.8)

floor(vec)
```

Figura 1.9: Documentação interna da função `round()`

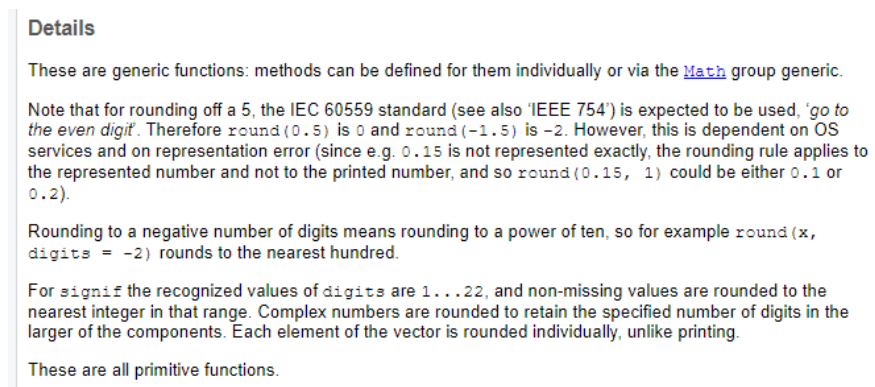
Fonte: Elaboração própria do autor.

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

```
ceiling(vec)
```

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

A seção de detalhes dessa documentação (mostrada na figura 1.10) é particularmente útil. Pois ela nos oferece uma boa descrição das implicações do padrão adotado pela função (IEC 60559). Além disso, a descrição presente na seção de detalhes também nos aponta uma particularidade importante sobre a função `round()`. Pois ao arredondar um decimal igual ao número 5, a função `round()` normalmente irá buscar o número inteiro par mais próximo.

Figura 1.10: Seção de detalhes da documentação interna da função `round()`

Fonte: Elaboração própria do autor.

A importância deste ponto, emerge do fato de que algumas funções de arredondamento muito utilizadas possuem um comportamento diferente de `round()`, em uma situação como essa. Um exemplo está na função `ARRED()` do Excel, que sempre arredonda um número para cima, a partir do momento em que a sua última casa decimal atinge um valor igual ou acima de 5. Se os nossos números são arredondados de formas distintas ao longo de certos programas, diferentes valores ou resultados podem ser estimados. Em geral, nós desejamos evitar isso.

Para que essa diferença fique clara, se eu arredondar os números 9,5 e 6,5, a função `round()` vai gerar como resultado, os números 10 e 6. Pois durante o processo de arredondamento, a função `round()` está preocupada em encontrar o número par mais próximo do valor em questão, e não sobre qual direção o arredondamento vai assumir.

```
vec <- c(9.5, 6.5, 4.5, 1.5, 2.5)
```

```
round(vec, digits = 0)
```

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

1.5.3 Ajuda Externa: referências, documentação oficial e canais úteis

Apesar de útil, a documentação interna de uma função é limitada. Essa situação tende a se confirmar especialmente em pacotes externos aos pacotes básicos do R, ao encontrarmos em suas documentações, seções de `Details` rasas e de pouca utilidade. Por isso, é interessante se aprofundar e conhecer outras referências externas ao R, produzidas por autores/usuários (livros-texto, cursos online, etc) que oferecem o seu conhecimento sobre a linguagem como um suporte à comunidade.

Ao longo desse livro, vamos descrever diversas funções que provêm dos pacotes do `tidyverse`. Por isso, é interessante que você se familiarize com os sites desses pacotes¹. Uma outra fonte rápida de informação, são as “colas” produzidas pela equipe do RStudio, chamadas de [RStudio Cheatsheets](#).

Além disso, temos diversos livros-textos importantes sobre a linguagem, que oferecem diversos conhecimentos extremamente valiosos, como as obras de [Wickham e Grolemund \(2017\)](#), [Gillispie e Lovelace \(2017\)](#), [Peng \(2015\)](#), [Grolemund \(2014\)](#), [Chambers \(2008\)](#), [Adler \(2010\)](#), além da documentação oficial da linguagem presente em [R Core Team \(2020a\)](#), [R Core Team \(2020b\)](#).

Também há diversos cursos e materiais disponíveis, que podem ser boas fontes de informação. Dentre eles, temos o curso [Introduction to R](#), da plataforma Datacamp. Além disso, temos um bom material de consulta em português, construído pela equipe do [Curso-R](#), além do material produzido pelo professor Walmes Marques Zeviani, intitulado [Manipulação e Visualização de Dados](#).

Para mais, temos alguns blogs que fazem boas reflexões e sempre trazem um bom conteúdo sobre a linguagem. Esse é o caso do site [R-Bloggers](#), que possui uma boa discussão sobre os mais diversos assuntos no R. Um outro exemplo, é o [blog do Tidyverse](#), que constantemente descreve novos

¹<https://www.tidyverse.org/packages/>

pacotes, novas funções disponíveis e novas aplicações para o R que podem ser muito interessantes para o seu trabalho.

Além dessas referências, é muito importante que você se familiarize com os canais de dúvida disponíveis, como o [Stackoverflow](#). Pois esses canais serão, com certeza, a sua principal fonte de ajuda no R. Em síntese, o StackOverflow funciona da seguinte maneira: 1) alguém envia uma pergunta; 2) cada pergunta, é marcada por um conjunto de *tags*, que definem a linguagem de programação, ou pacote, ou assunto específico a que se refere a dúvida; 3) qualquer pessoa, pode postar uma resposta nessa pergunta, ou algum comentário que seja útil; 4) as respostas mais úteis e completas, serão votadas para cima, pelos próprios usuários do site; 5) dessa forma, as respostas mais úteis e completas, vão sempre aparecer primeiro na postagem da dúvida em questão.

Para encontrar perguntas especificamente voltadas para a linguagem R no StackOverflow, você deve sempre procurar por perguntas marcadas com a *tag* [r], ou por algum pacote específico da linguagem. Por exemplo, o StackOverflow contém um estoque enorme de dúvidas marcadas com a *tag* ggplot2, que se refere ao pacote ggplot2, que vamos discutir mais a frente. Logo, o StackOverflow representa uma fonte extremamente importante sobre esse pacote.

Além do StackOverflow, nós também possuímos o [RStudio Community](#), que também é um canal de dúvidas bastante ativo, e que funciona de maneira muito similar ao StackOverflow. Onde pessoas fazem uma pergunta, que é marcada por *tags* que definem o pacote ou o assunto específico que a pergunta se refere. Porém, as perguntas no RStudio Community, tendem a assumir um aspecto mais parecido com uma discussão (ao invés de um caráter de pergunta-resposta presente no StackOverflow). Ou seja, uma pergunta abre de certa forma, uma discussão. Uma pessoa fornece uma resposta, depois outra fornece um outro olhar sobre a pergunta, o autor descreve novas dúvidas, novas respostas surgem, podendo assim, criar uma discussão infundável em torno da dúvida inicial.

As fontes de ajuda externas ao R, serão a sua maior ajuda, e a sua principal referência. Pois como foi destacado por [Chase \(2020\)](#), ninguém é completamente autodidata. Todos nós cometemos erros, e uma das grandes vantagens de uma comunidade como a do R, é que muito conhecimento é produzido e compartilhado em torno desses erros. Por essa razão, [Chase \(2020\)](#), assim como os autores desta obra, preferimos nos caracterizar como seres instruídos pela comunidade (*community-taught*).

1.5.4 Um exemplo clássico de ajuda externa

Uma das primeiras dúvidas que atingem os iniciantes, diz respeito aos objetos criados, ou melhor dizendo, aos objetos não criados em sua sessão. Você já viu na seção [Introdução a objetos](#), qual a estrutura básica necessária para criarmos um objeto (`nome_do_objeto <- valor_do_objeto`). Porém, você pode acabar se perdendo durante o seu trabalho, de forma a não saber quais objetos você criou em sua sessão. Em situações como essa, você pode executar a função `ls()`. Essa função irá listar o nome de todos os objetos que estão criados em sua sessão atual do R.

```
ls()
```

```
## [1] "casa"           "cAsa"           "def.chunk.hook"
```

```
## [4] "idade_pedro"      "input"      "load_packages"
## [7] "numero_importante" "pckgs"      "soma"
## [10] "vec"            "vec_num"
```

Com isso, caso você esteja em dúvida se você já criou ou não, um certo objeto em sua sessão, você pode conferir se o nome deste objeto aparece nessa lista resultante da função `ls()`. Caso o nome do objeto não se encontre nela, você sabe que o objeto em questão ainda não foi criado.

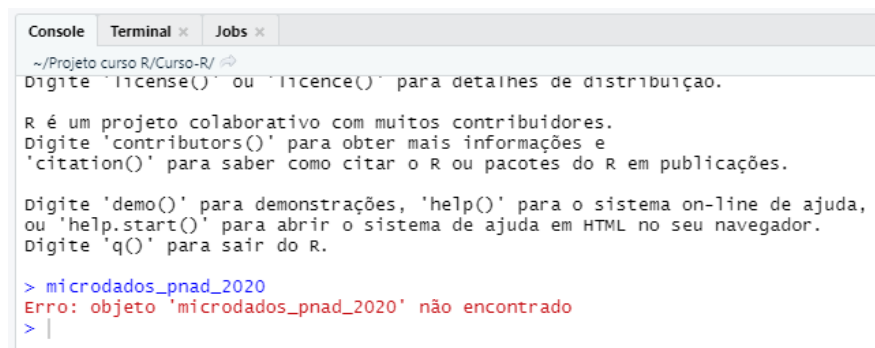
Por outro lado, você pode estar interessado em apagar um certo objeto de sua sessão. Tal resultado, pode ser atingido através da função `rm()`. Logo, se eu possuo um objeto chamado `dados_2007`, e eu desejo eliminá-lo de minha sessão, eu preciso apenas fornecer o nome deste objeto à função `rm()`.

```
# Removendo o objeto chamado
# dados_2007 de minha sessão
rm(dados_2007)
```

Na próxima seção, vamos abordar o uso de *scripts* no R, e um erro muito comum quando se está iniciando com os *scripts*, é o de se esquecer de efetuar os comandos para criar um objeto. Ou seja, muitos iniciantes escrevem no *script*, os comandos necessários para criar o seu objeto, mas acabam se esquecendo de enviar esses comandos para o Console, onde eles serão avaliados e executados.

A função `ls()` oferece uma forma rápida de consulta, que pode sanar a sua dúvida em ocasiões como essa. Mas uma outra forma ainda mais efetiva de sanarmos essa dúvida, consiste em chamar pelo nome deste objeto no console. Se algum erro for retornado, há grandes chances de que você ainda não criou esse objeto em sua sessão. Veja o exemplo abaixo na figura 1.11, em que chamo por um objeto chamado `microdados_pnad_2020`, e um erro é retornado.

Figura 1.11: Mensagem de erro - Console



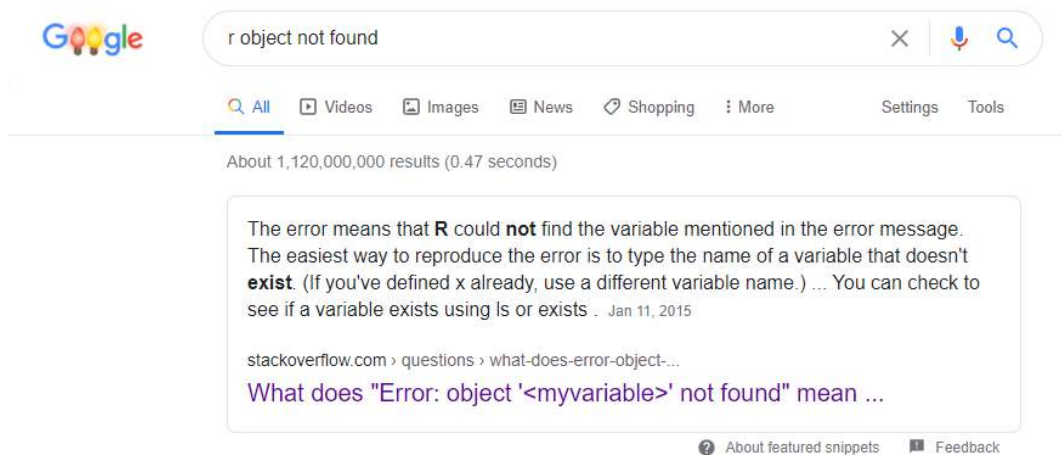
Fonte: Elaboração própria do autor.

Sempre que você não souber o que um erro significa, ou a que termo ele está se referindo, faça uma pesquisa rápida sobre esse erro no Google. Se o seu erro está sendo gerado, ao executar uma função específica, você pode anteceder o erro gerado, por um “R” e pelo nome da função utilizada,

na barra de pesquisa do Google. No nosso caso, talvez seja melhor pesquisarmos apenas pelo erro antecedido por um “R”, como na figura abaixo. Há alguma chance de você encontrar referências de ajuda em português. Porém, as chances são infinitamente maiores se você pesquisar por artigos e perguntas escritas em inglês. Por isso, se a sua mensagem de erro estiver em português (como é o caso da mensagem na figura acima), é melhor que você tente traduzí-la para o inglês, caso você tente pesquisar por ela no Google.

Podemos encontrar no primeiro link da página mostrada na figura 1.12, uma pergunta postada no StackOverflow. Como o StackOverflow é geralmente uma boa referência de ajuda, há uma boa chance de encontrarmos o que estamos necessitando nesse link.

Figura 1.12: Pesquisa Google sobre mensagem de erro



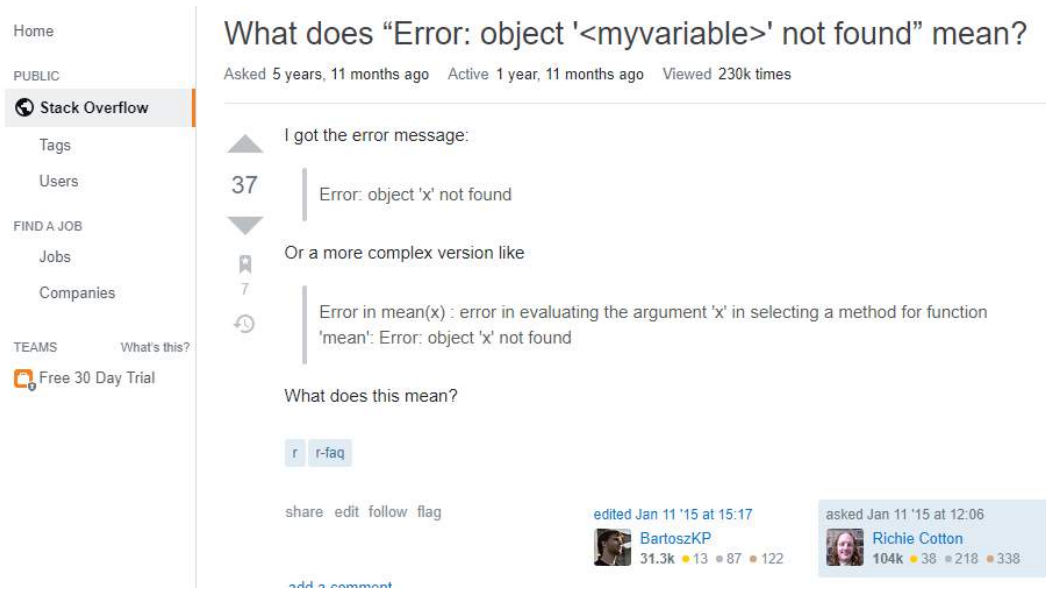
Fonte: Elaboração própria do autor.

Ao acessarmos uma pergunta do StackOverflow, a primeira parte que aparece em sua tela, é a pergunta em si. Perguntas que são muito úteis, e que traduzem uma dúvida muito comum dos usuários, tendem a ser “votadas para cima”. A pergunta exposta na figura 1.13, possui 37 votos, o que indica ser uma pergunta comum e útil o suficiente para ajudar no mínimo 37 pessoas.

Logo abaixo da pergunta em si, temos as respostas de usuários que se dispuseram a respondê-la. As respostas mais úteis para a pergunta em questão, tendem a ter maiores votos dos usuários e, por isso, tendem a aparecer primeiro na página em relação as outras respostas menos úteis. Como podemos ver na figura 1.14, a primeira resposta possui 33 votos.

A resposta mostrada na figura 1.14, é bem esclarecedora. Como o autor pontua, um erro do tipo “objeto x não foi encontrado” (ou “*object x not found*”) ocorre quando tentamos utilizar um objeto que ainda não existe, um objeto que ainda não foi definido. A partir do momento em que voce definir esse objeto, este erro não ocorre mais.

Como pontuei anteriormente, é muito comum de um aluno escrever os comandos necessários para

Figura 1.13: Pergunta StackOverflow - Parte 1

Fonte: Elaboração própria do autor.

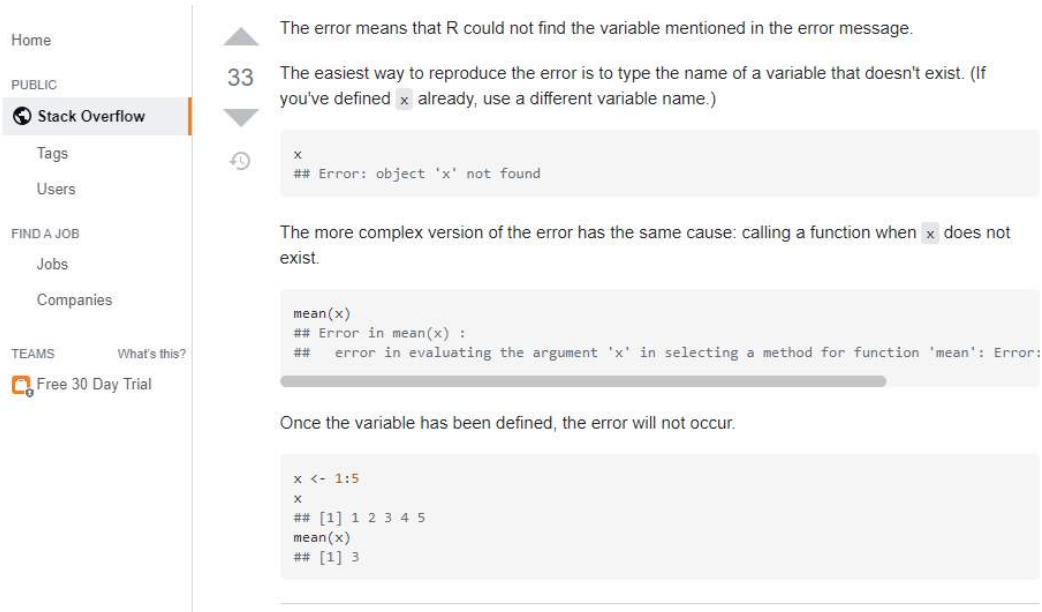
criar um objeto em seu *script*, mas se esquecer de enviar esses comandos do *script* para o Console, onde serão avaliados e executados. Por isso, sempre que ocorrer esse erro, confira se você conseguiu enviar os comandos para o Console. Também confira se os comandos utilizados para criar o objeto, foram de fato executados, isto é, confirme se nenhum erro apareceu durante a execução desses comandos. Pois a depender da gravidade do erro gerado, a execução dos comandos pode ter sido comprometida e, portanto, o objeto não pôde ser criado.

Por isso, sempre que enfrentar algum erro no R, tente fazer uma pesquisa rápida no Google. Em geral, você pode copiar e colar diretamente a mensagem, ou citar apenas trechos, ou a oração principal da mensagem de erro na pesquisa. É interessante sempre colocar um “r” antes da mensagem de erro, para definir um pouco melhor a sua pesquisa e encontrar links referentes à linguagem R. Uma boa referência externa para compreender e solucionar erros no R, é o StackOverflow.

1.6 Scripts

Até o momento, estivemos utilizando diretamente o Console para executarmos os nossos comandos. Porém, você provavelmente se sentiu um pouco perdido ao procurar os últimos comandos que você executou no console e, se sentiu um pouco frustrado ao ter de digitar novamente o comando caso queira executá-lo uma segunda vez. Por essa razão, a medida que você trabalha com o R, a necessidade de guardar os seus comandos anteriores em algum lugar, se torna cada vez mais urgente. Para isso, você pode utilizar um *script*.

Figura 1.14: Pergunta StackOverflow - Parte 2



Fonte: Elaboração própria do autor.

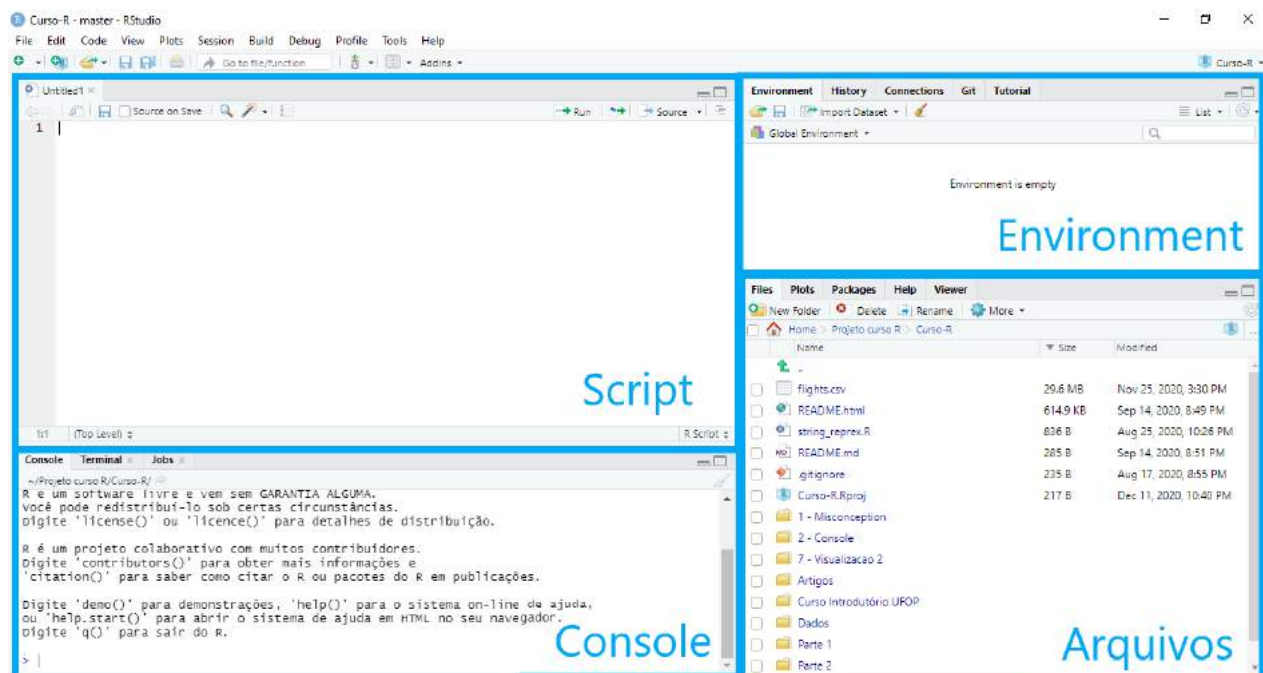
Um *script* é um simples arquivo de texto, que contém a extensão `.R`, para indicar que todo o texto contido neste arquivo, representam comandos do R. Portanto, um *script* contém um conjunto de códigos e comandos do R que podem ser facilmente acessados, editados e executados através das ferramentas e atalhos do RStudio, tornando o seu fluxo de trabalho com o R mais eficiente. Ao utilizar o RStudio, os códigos contidos nos scripts podem ser executados individualmente ou em conjunto.

Para criar um script no RStudio, você possui duas opções: 1) clicar em *File* → *New File* → *R Script*; ou 2) utilizar o atalho `Ctrl + Shift + N`. Após criar o *script*, o quadrante esquerdo do RStudio será dividido verticalmente em dois: a parte superior comporta o editor de *script*'s e a inferior o Console. Como resultado, o seu ambiente do RStudio ficará semelhante ao ambiente exibido na figura 1.15.

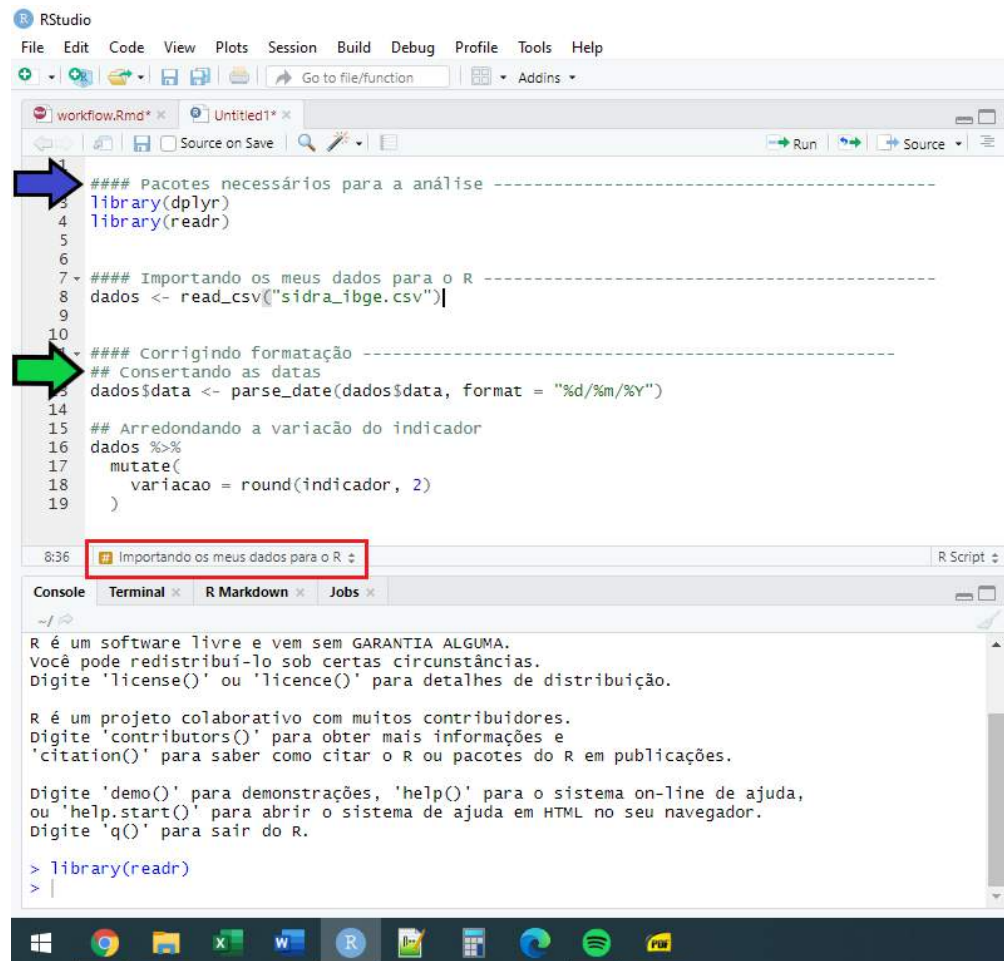
Você pode criar títulos que delimitam as áreas, ou as etapas de seu *script*, e é uma forma muito eficiente de navegar pelo seu *script*, caso ele seja muito grande. Na figura 1.16, um exemplo destes títulos está identificado pela seta azul. Também na figura 1.16, temos uma caixa vermelha, e dentro dela podemos ver uma referência que aponta qual a seção, ou melhor, qual o título da seção no qual o nosso cursor se encontra atualmente. O meu cursor se encontra no momento, na seção “Importando os dados para o R”. Ao clicar sobre esta referência especificada na caixa vermelha, uma nova caixa de seleção irá aparecer contendo cada um dos títulos que você criou em seu *script*, e ao clicar sobre um destes títulos, você será redirecionado para o início desta seção no *script*.

Esses títulos especiais, são formados pela união entre o caractere de comentário do R (`#` - *hashtag*), o texto que você quer inserir neste título, e vários sinais de menos (`-`) em sequência, formando assim

Figura 1.15: Quadrantes da área de trabalho do RStudio, após a abertura de um script



Fonte: Elaboração própria do autor.

Figura 1.16: Títulos e comentários em scripts

Fonte: Elaboração própria do autor.

a seguinte estrutura: `### <título desejado> -----`. O número de *hashtag*'s e de sinais de menos que você insere, são arbitrários. Ao invés de escrevê-los a mão, o RStudio oferece um atalho que cria automaticamente esses títulos, através das teclas `Ctrl + Shift + R`.

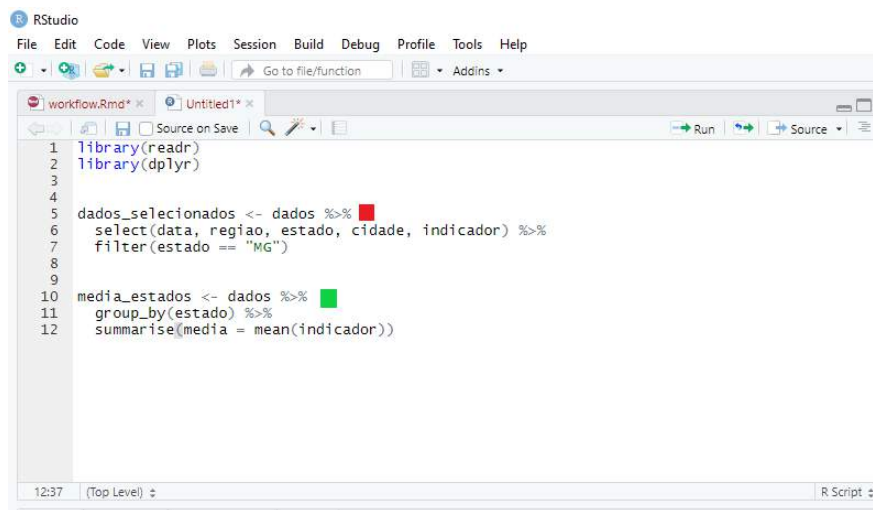
Lembre-se que você também pode adicionar pequenas anotações e comentários em seu *script* com *hashtags* (`#`). Nós definimos em seções anteriores, que este é um caractere especial da linguagem, e que qualquer texto que você colocar a frente dele, será ignorado pelo console. Na figura 1.16, temos um exemplo deste comentário que está marcado por uma seta verde.

Esses comentários são uma boa forma de descrever o que os comandos abaixo dele fazem, ou então de apontar configurações e cuidados importantes que você deve ter com esses comandos. Isso é importante especialmente com aquelas funções que você raramente utiliza, pois é menos provável que você se lembre de como elas funcionam, ou de como elas se comportam.

1.6.1 Executando comandos de um script

A essa altura, você já sabe que para executarmos qualquer comando do R, ele precisa ser enviado para o console, onde será avaliado e executado. Por isso, ao utilizarmos um *script*, desejamos uma forma rápida de enviarmos esses comandos que estão guardados neste *script*, para o console do R. O RStudio oferece um atalho para isso, que é o `Ctrl + Enter`. Veja a figura 1.17, se o cursor de seu mouse estiver sobre o retângulo vermelho desenhado no *script*, ao apertar o atalho `Ctrl + Enter`, o RStudio enviará todo o bloco de comandos que criam o objeto `dados_selecionados`, para o console. Agora, se o cursor de seu mouse estivesse sobre o retângulo verde desenhado no *script*, o RStudio enviaria o bloco de comandos que formam o objeto `media_estados`.

Figura 1.17: Executando comandos de um script



Fonte: Elaboração própria do autor.

Após enviar um bloco de comandos para o console, através deste atalho, o RStudio irá automati-

camente mover o cursor de seu mouse para o próximo bloco de comandos. Desta maneira, você pode executar parte por parte de seu *script* em sequência e, conferir os resultados de cada bloco no console.

Além disso, o RStudio também oferece um outro atalho para caso você queira executar todos os comandos de um *script* de uma vez só. Para isso, você pode apertar as teclas `Ctrl + Alt + R`.

1.6.2 Salvando um *script*

Ao salvar o seu *script*, você está salvando os comandos necessários para gerar os seus resultados. Isto é, através de *script*'s você possui uma poderosa ferramenta para a reproducibilidade de sua análise. Em outras palavras, com um *script*, você é capaz de salvar **os comandos necessários para se obter o resultado desejado**, no lugar dos próprios resultados em si. Dito de outra forma, é muito mais prático carregarmos a metodologia necessária para se obter um resultado, do que o resultado em si. Pois você pode gerar repetidamente os mesmos resultados através dos comandos salvos em seu *script*, quantas vezes forem necessárias. Por outro lado, você não é capaz de gerar o *script*, ou os comandos necessários, ou a metodologia de cálculo utilizada, a partir de seus resultados.

Para salvar um *script* que está aberto em seu RStudio, você pode clicar em *File* → *Save As...*, e escolher o diretório em que o arquivo será guardado. Você também pode salvar esse *script*, ao clicar sobre o símbolo de disquete, presente logo abaixo do nome desse *script*, no canto superior direito. Uma vez definido o nome do *script* e o local onde ele será guardado, você pode clicar em *File* → *Save*, ou utilizar o atalho `Ctrl + S` para salvar o *script* corrente a medida em que você for editando ele.

Além desses pontos, lembre-se que um *script* é nada mais do que um arquivo de texto com uma extensão `.R` e, por isso, ele pode ser aberto normalmente por editores de texto padrão (como o Bloco de Notas do Windows, ou por programas como Notepad ++ e Sublime Text).

1.7 Pacotes

Como descrevemos anteriormente na seção **O sistema e universo do R**, o R pode ser dividido em duas partes: os pacotes básicos da linguagem; e todos os demais pacotes externos que foram criados e ofertados pela comunidade do R. Um pacote (*package*) corresponde a unidade fundamental de compartilhamento de códigos e funções no R (WICKHAM, 2015b). Dito de outra forma, segundo as palavras de Wickham e Grolemund (2017), um pacote do R é uma coleção de funções, dados e documentação que estendem as funcionalidades do R.

No momento de escrita desta obra (novembro de 2020), existem mais de **16.000 pacotes disponíveis no CRAN**. Segundo Wickham (2015b), esta grande variedade de pacotes representa uma das principais razões para o sucesso do R nos anos recentes, e ressalta o seguinte pensamento: **é bastante provável que algum usuário já tenha enfrentado o mesmo problema que você, e após solucioná-lo, tenha ofertado um pacote que possa auxiliar você, na busca dessa solução**. Logo,

you can obtain enormous benefits by using the set of functions developed by other users to solve their problems.

1.7.1 Como utilizar um pacote

Como é descrito por [Adler \(2010\)](#), para utilizarmos um pacote no R, precisamos “carregá-lo” para a nossa sessão. Porém, para “carregarmos” um pacote para a nossa sessão, esse pacote precisa estar instalado em nosso computador. Logo, em resumo, nós devemos realizar os seguintes passos ²:

1. Instalar o pacote a partir do servidor do CRAN: `install.packages("nome_do_pacote")`.
2. Carregar o pacote em cada sessão no R: `library(nome_do_pacote)`.

Você precisa executar o primeiro passo (instalar o pacote com a função `install.packages()`) apenas uma vez. Após instalar o pacote em sua máquina, você precisa carregar esse pacote através da função `library()` em toda sessão no R que você deseja utilizar as funções desse pacote. Ou seja, toda vez que iniciar o R, você precisa carregar o pacote para ter acesso às suas funções.

Por exemplo, se você desejasse utilizar as funções disponíveis no pacote `ggplot2`, que possui um conjunto de funções voltadas para a composição de gráficos, você precisaria dos comandos abaixo. Repare que o nome do pacote é fornecido como *string* à função `install.packages()`. Logo, sempre que for instalar um pacote, lembre-se de contornar o nome do pacote por aspas (simples ou duplas).

```
# Instalar o pacote `ggplot2` em seu computador
install.packages("ggplot2")
# Carregar o pacote `ggplot2` em sua sessão atual do R
library(ggplot2)
```

Como [Gillespie e Lovelace \(2017\)](#) destaca, uma boa prática a ser adotada é carregar todos os pacotes necessários sempre no início de seu *script*. Dessa forma, você está acoplando a sua sessão, todas as dependências necessárias para aplicar todas as funções dispostas ao longo de seu *script*.

1.7.2 Identificando os pacotes instalados em sua máquina e aqueles que foram carregados para a sua sessão

Um dos métodos mais diretos de se identificar se um determinado pacote está ou não carregado em sua sessão, consiste em você tentar utilizar uma das funções desse pacote. Se um erro aparecer durante esse processo, indicando que tal função não foi encontrada ou que ela não existe, há grandes chances de que o pacote pelo qual você está preocupado, não se encontra disponível em sua sessão atual.

²Uma parte pequena dos pacotes disponíveis, não se encontram no CRAN, mas sim em outras plataformas como o GitHub. Neste caso, você precisa instalá-los a partir de funções do pacote `devtools`. Para mais detalhes, consulte o item [Installing a Package from GitHub](#) de Long e Teetor (2019).

Por exemplo, eu posso tentar utilizar a função `mutate()` do pacote `dplyr` como eu normalmente faria. Pela mensagem de erro abaixo, sabemos que o R não pôde encontrar a função `mutate()`, logo, o pacote `dplyr` provavelmente não foi carregado para a minha sessão até o momento.

```
mutate()
```

```
Error in mutate() : não foi possível encontrar a função "mutate"
```

Apesar de rápido, este método é um pouco inseguro. Pois talvez um dos pacotes que já estão carregados em minha sessão, possua uma função com o nome `mutate()`. Em outras palavras, ao tentar rodar a função `mutate()` em minha sessão, pode ser que o R encontre uma função `mutate()` diferente da que estou procurando. Por isso, um método mais seguro é necessário.

A resposta para tal necessidade se encontra na lista de *environments* conectados. Cada pacote carregado para a sua sessão, é representado por um *environment* que está acoplado ao seu *environment* principal. Logo, ao descobirmos todos os *environments* presentes em nossa sessão, nós podemos identificar todos os pacotes que foram carregados. Para obtermos uma lista dos *environments* presentes em nossa sessão, nós podemos executar a função `search()`, como abaixo:

```
search()
```

```
## [1] ".GlobalEnv"          "package:RcppRoll"      "package:nycflights13"
## [4] "package:SAScii"      "package:haven"         "package:readxl"
## [7] "package:vctr"        "package:lobstr"        "package:lubridate"
## [10] "package:rmarkdown"  "package:bookdown"      "package:knitr"
## [13] "package:glue"        "package:fs"            "package:magrittr"
## [16] "package:forcats"     "package:stringr"       "package:dplyr"
## [19] "package:purrr"       "package:readr"         "package:tidyr"
## [22] "package:tibble"     "package:ggplot2"       "package:tidyverse"
## [25] "package:yaml"       "tools:rstudio"         "package:stats"
## [28] "package:graphics"   "package:grDevices"     "package:utils"
## [31] "package:datasets"   "package:methods"       "Autoloader"
## [34] "package:base"
```

Os valores que estiverem na forma `package:nome_do_pacote` indicam o *environment* de um pacote que está carregado em sua sessão atual do R. Já o valor denominado `.GlobalEnv`, representa o *global environment*, que é o seu *environment* principal de trabalho, onde todos os seus objetos criados são salvos. Os *environments* no R, representam os “espaços”, ou “ambientes” onde os seus objetos são guardados. Portanto, os objetos que você cria em sua sessão no R, são guardados nesse *environment* denominado `.GlobalEnv`. Enquanto isso, todas as funções e objetos disponíveis, por exemplo, no pacote `tibble`, estão guardados no *environment* chamado `package:tibble`. Vamos descrever em mais detalhes esses pontos, na seção [Noções básicas de environments](#).

Por outro lado, você talvez enfrente algum erro ao tentar carregar o pacote de seu interesse. Nesse caso, um bom movimento seria se certificar que esse pacote está instalado em sua máquina. Segundo

Adler (2010), se você precisa identificar todos os pacotes instalados em sua máquina, você pode executar a função `library()` sem definir nenhum argumento ou pacote em específico.

```
# Uma nova janela será aberta em seu RStudio
# contendo uma lista de todos os pacotes instalados
library()
```

1.7.3 Acessando as funções de um pacote sem carregá-lo para sua sessão

Apesar de ser uma prática ideal na maioria das situações, você talvez não queira carregar um pacote específico e, mesmo assim, utilizar uma de suas funções. Tal opção pode gerar uma importante economia de espaço em sua memória RAM, durante a sua análise. Até porque, se você irá utilizar apenas uma função do pacote, talvez não haja necessidade de carregar o pacote inteiro.

Para acessarmos uma função de um pacote que não foi carregado ainda em nossa sessão, precisamos chamar primeiro pelo pacote de onde estamos tirando a função, como na estrutura abaixo.

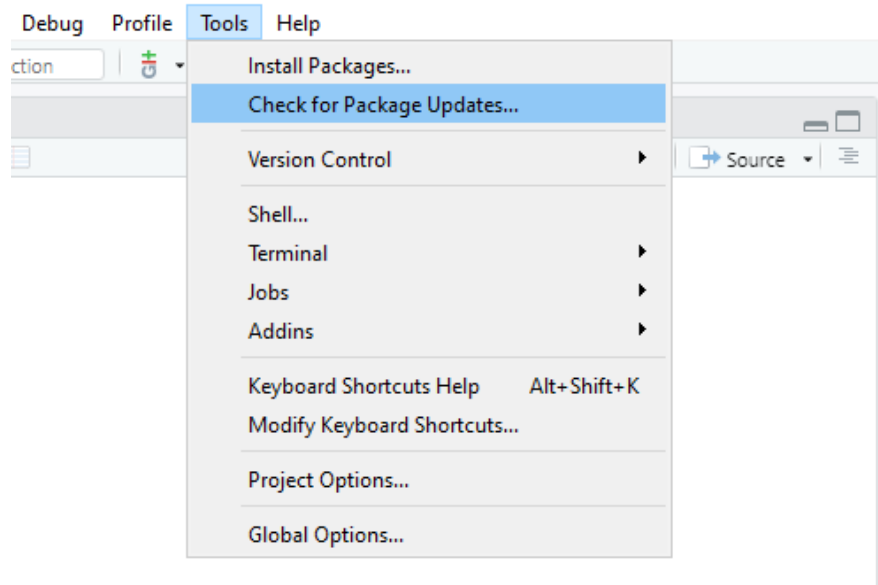
```
# Acessar uma função de um pacote sem carregá-lo
nome_do_pacote::nome_da_função()
```

Logo, se você quisesse acessar a função `filter()` do pacote `dplyr`, por exemplo, você precisa primeiro chamar pelo pacote `dplyr` e, em seguida, posicionar duas vezes dois pontos (`:`) para acessar uma função ou objeto presente neste pacote. Por último, basta digitar o nome da função de interesse.

```
# Para acessar a função filter() sem chamar
# pelo pacote dplyr
dplyr::filter()
```

1.7.4 Atualizando pacotes

A linguagem R está o tempo todo evoluindo e se aprimorando e, por essa razão, muitos dos pacotes disponíveis hoje, são constantemente atualizados, com o objetivo de implementar novas funcionalidades e/ou aperfeiçoar a eficiência de suas funções. Logo, é uma boa prática que você mantenha os pacotes instalados em seu computador, constantemente atualizados. Para atualizar um pacote, você precisa apenas instalá-lo novamente, através da função `install.packages("nome_do_pacote")`, ou acessar a opção *Tools* → *Check for Packages Updates...* no RStudio, como está demonstrado na figura 1.18. Através dessa opção, o RStudio irá listar todos os pacotes que possuem versões mais recentes e, portanto, podem ser atualizados. A grande vantagem é que você pode atualizar todos os pacotes presentes nessa lista de uma vez só.

Figura 1.18: RStudio: Opção para atualização de pacotes

Fonte: Elaboração própria do autor.

Exercícios

Exercício 1

Tente calcular algumas operações básicas:

- A) Qual é a soma entre 32397 e 55405?
- B) Calcule a soma total do conjunto de valores dispostos no vetor `conj` abaixo.

```
conj <- c(290, 34, 512, 54, 89, 10)
```
- C) Considerando que, $y = 3x^3 - 12x^2 + \frac{1}{15}x + 25$, calcule o valor de y quando x é igual a 5.

Exercício 2

Em cada item abaixo, temos uma mensagem de erro específica que supostamente apareceu em seu console. Tente explicar como ou porque essas mensagens podem aparecer em seu console. Em outras palavras, tente explicar o que essas mensagens significam, e qual foi o fato ocorrido que gerou esses erros:

- A) Erro: objeto 'logica' não encontrado.
- B) Error in `bind_rows()` : não foi possível encontrar a função "bind_rows"

C) Error in library(dplyr) : there is no package called ‘dplyr’

Exercício 3

As próximas questões vão implicitamente esperar que você utilize algumas dessas funções: `sum()`, `mean()`, `abs()` e `sd()`. Claro que, o R te oferece a liberdade de escrever as suas próprias funções, ou de desenhar o seu próprio caminho até as soluções dessas questões. Portanto, não se preocupe se você encontrar uma solução para as questões abaixo, que não incluem o uso dessas funções específicas.

- A) Considerando que a variável X segue uma distribuição normal, como você normalizaria (isto é, calcular o índice Z da distribuição) os valores presentes no vetor `vec` abaixo, que contém uma amostra de valores da variável X .

```
vec <- c(0.5, 1.2, 2.5, 1.3, 2.2, 3.7)
```

- B) Utilizando novamente a variável `vec`, calcule o seu desvio médio.

```
vec <- c(0.5, 1.2, 2.5, 1.3, 2.2, 3.7)
```

Capítulo 2

Fundamentos da Linguagem R

2.1 Introdução

Nas próximas seções vou abordar os fundamentos da linguagem: os básicos de sua sintaxe, quais são as estruturas e tipos de dados que a linguagem oferece, e como as suas regras de *coercion* funcionam.

Na maior parte de sua análise, você não vai estar interessado em como o R está estruturando ou interpretando os seus dados em um dado momento. Porém, várias das funções ou ações que você deseja aplicar, exigem que os seus dados estejam estruturados em uma forma específica. Logo, ter familiaridade com os fundamentos do R, com as suas estruturas e suas propriedades, e principalmente, poder reconhecê-las, vai te salvar muito tempo. Com esse conhecimento, será mais fácil de você evitar erros, e será mais fácil de identificar e transformar a estrutura de seus dados para qualquer que seja a sua necessidade em um dado momento de sua análise.

Tendo isso em mente, além de introduzir a linguagem, as próximas seções também tem como objetivo, lhe fornecer um base sólida desses fundamentos, para que você possa identificar e transitar entre essas diversas estruturas e tipos de dados, de forma flúida.

2.2 Objetos (uma revisão)

Uma das principais características do R, é que ele é uma linguagem orientada a objetos (*object oriented*). Isto significa, que quando você estiver trabalhando com seus dados no R, você estará aplicando operações e transformações sobre os objetos onde seus dados estão guardados.

Os objetos no R, são como as caixas que você utiliza na sua mudança. Você guarda algo dentro dessa caixa, e coloca um adesivo com um nome para essa caixa, para que você se lembre do que está dentro dela. No dia seguinte à mudança, quando você precisar do conteúdo que está guardado naquela caixa, você procura essa caixa pelo nome que você deu a ela.

No exemplo abaixo, eu estou criando um objeto, que dou o nome de `data_aniversario`, e estou

utilizando o símbolo `<-` para definir o valor deste objeto para a data de aniversário de um amigo importante (20 de maio). O símbolo `<-` é comumente chamado de *assignment*, e significa que estamos atribuindo um valor a um objeto (no caso abaixo, `data_aniversario`). Em outras palavras, os comandos abaixo, podem ser lidos como: eu atribuo ao objeto de nome `data_aniversario`, o valor de "20 de maio". Após isso, sempre que eu chamar por esse nome, o R irá procurar por uma caixa (ou um objeto) que possui um adesivo com um nome de `data_aniversario`. Quando ele encontrar essa caixa, ele irá me retornar no console o que tem dentro dessa caixa (ou desse objeto).

```
data_aniversario <- "20 de maio"

### Quando eu chamo pelo nome deste objeto
### no console, o R me retorna o que tem dentro dele.
data_aniversario

## [1] "20 de maio"
```

O conceito de objeto é uma metáfora, ou uma forma útil de enxergarmos este sistema. Pois para o R, o nome `data_aniversario` se trata apenas uma conexão até o valor ("20 de maio"). Para demonstrarmos essa ideia, vamos utilizar os endereços desses objetos. Isto é, todos os valores contidos nos objetos que você cria em sua sessão do R, vão obrigatoriamente ocupar um espaço, ou um endereço da memória RAM de seu computador. Enquanto este objeto estiver “vivo”, ou seja, enquanto esta conexão entre o nome `x` e os seus valores permanecer acessível em sua sessão, esses valores vão estar ocupando um endereço específico de sua memória RAM. Para descobrirmos esse endereço, nós podemos utilizar a função `ref()` do pacote `lobstr`. Vamos supor por exemplo, que nós criamos um vetor chamado `x`, que contém três números. Perceba abaixo pelo resultado da função `ref()`, que ao criar este objeto `x`, os seus valores foram alocados no endereço `0x1ca169c03d8` da minha memória RAM.

```
library(lobstr)

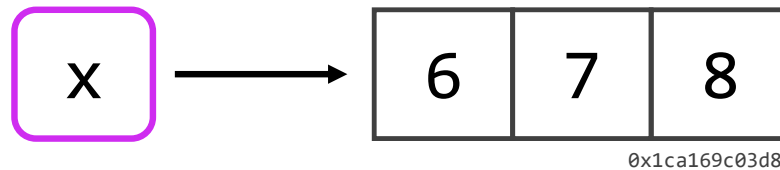
x <- c(6, 7, 8)

ref(x)

## [1:0x1ca169c03d8] <dbl>
```

Portanto, um objeto no R, nada mais é do que uma conexão entre um nome e valores que estão guardados em um endereço da memória RAM de seu computador. Os únicos momentos em que este endereço muda, serão todas as vezes em que você reiniciar a sua sessão no R, ou todas vezes em que você executar novamente os códigos necessários para criar os seus objetos. Tendo isso em mente, em uma representação visual, um objeto no R pode ser representado da seguinte maneira:

Para desenvolvermos essa ideia, pense o que ocorreria, se atribuíssemos os valores do objeto `x`, a um novo objeto. Segundo essa perspectiva, nós estaríamos apenas conectando o vetor com os valores 6, 7 e 8, a um novo nome, no exemplo abaixo, ao nome `y`. Nós poderíamos utilizar novamente a

Figura 2.1: Representação de um objeto

Fonte: Elaboração própria do autor. Inspirado em [WICKHAM, 2015a](#), Cáp. 2.

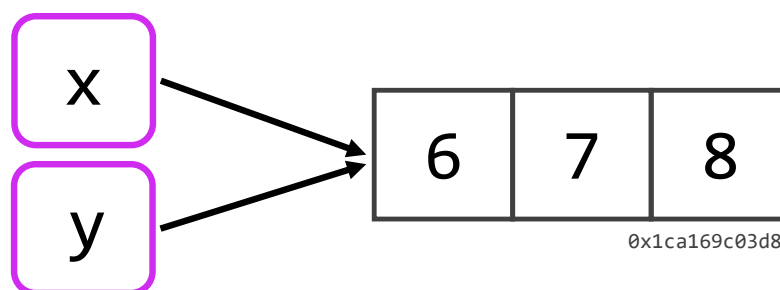
função `ref()` para conferirmos o endereço onde os valores do objeto `y`, se encontram, e perceba que eles estão no mesmo local que os valores do objeto `x`.

```
y <- x
```

```
ref(y)
```

```
## [1:0x1ca169c03d8] <dbl>
```

Logo, se atualizarmos a nossa representação visual, temos o seguinte resultado:

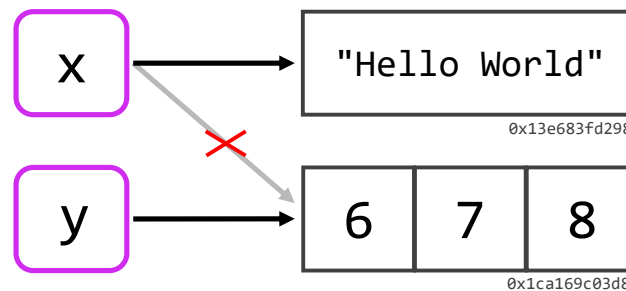
Figura 2.2: Conectando mais nomes a um mesmo conjunto de valores

Fonte: Elaboração própria do autor. Inspirado em [WICKHAM, 2015a](#), Cáp. 2.

Em outras palavras, o R em nenhum momento criou uma cópia do vetor contendo os valores 6, 7 e 8, e alocou essa cópia no objeto `y`. Ele apenas conectou um novo nome (`y`) a esse vetor de valores. Por isso, quando você possui um objeto, e atribui um novo valor a este objeto, você está na verdade eliminando a conexão que o nome deste objeto possuía com o valor que estava guardado anteriormente naquele objeto. Ou seja, se você retornar ao vetor `x`, e definir um novo valor para ele, você estaria eliminando a sua conexão com o vetor que contém os números 6, 7 e 8, e atribuindo

essa conexão a um outro conjunto de valores. Por exemplo, caso eu executasse o comando `x <- "Hello World"`, o resultado seria uma nova conexão como você pode ver pela figura 2.3.

Figura 2.3: Atribuindo novos valores a seus objetos



Fonte: Elaboração própria do autor. Inspirado em [WICKHAM, 2015a](#), Cáp. 2.

O R vai jogar fora, qualquer valor que não esteja conectado a um nome, ou a um objeto em sua sessão. Logo, tendo em mente a figura 2.3, caso eu atribuísse um novo valor ao objeto `y`, uma outra conexão até o vetor que contém os números 6, 7 e 8, seria eliminada. Com isso, este vetor não possuiria mais nenhuma conexão até um nome, e por isso, seria descartado pelo R. Portanto, se você precisa atribuir um novo valor para um objeto, mas deseja manter o valor que você deu a ele anteriormente, basta que você crie uma nova conexão até o valor antigo. Em outras palavras, se você quer manter este valor, basta conectá-lo a um novo objeto.

No exemplo abaixo, eu crio um objeto (`economista_1`) contendo o nome de um economista famoso, e em seguida conecto este nome a um novo objeto (`economista_anterior`). Portanto, o nome de Keynes está agora conectado a dois nomes, ou está contido em dois objetos diferentes em sua sessão no R. Por último, eu sobreponho o nome de Keynes que guardei no primeiro objeto (`economista_1`), pelo nome de outro economista famoso. Quando faço isso, estou efetivamente eliminando uma das conexões até o nome de Keynes, e atribuindo essa conexão ao nome de Schumpeter. Porém, como o nome de Keynes ainda possui uma conexão existente (`economista_anterior`), o nome continua “vivo” e presente em nossa sessão, e se quisermos acessar novamente esse nome, basta chamarmos pelo objeto onde o salvamos.

```
# Primeiro valor
economista_1 <- "John Maynard Keynes"

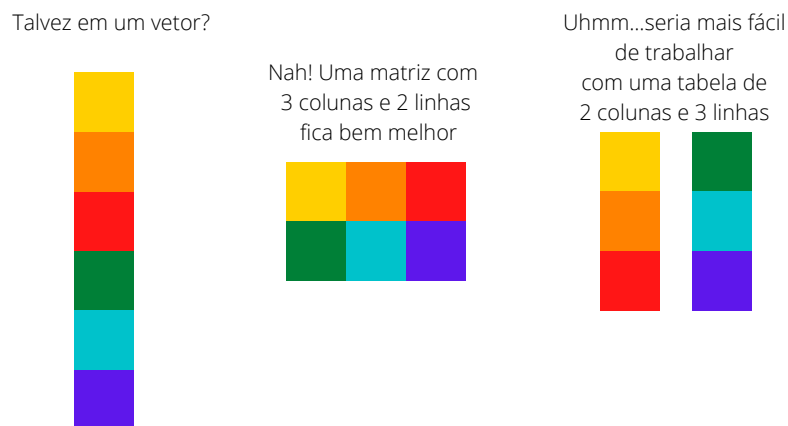
# Atribuindo o primeiro valor a um novo
# objeto
economista_anterior <- economista_1
```

```
# Sobrepondo o primeiro valor no  
# primeiro objeto com um novo nome  
economista_1 <- "Joseph Alois Schumpeter"  
  
economista_1  
  
## [1] "Joseph Alois Schumpeter"  
  
economista_anterior  
  
## [1] "John Maynard Keynes"
```

2.3 Estruturas e tipos de dados

O R possui diferentes formas de estruturar (ou organizar) os dados que você fornece a ele. Essas formas são o que estou chamando de estruturas de dados. Quando estamos decidindo em qual estrutura devemos guardar os nossos dados, estamos basicamente fazendo o processo descrito na figura 2.4:

Figura 2.4: Estruturas de dados



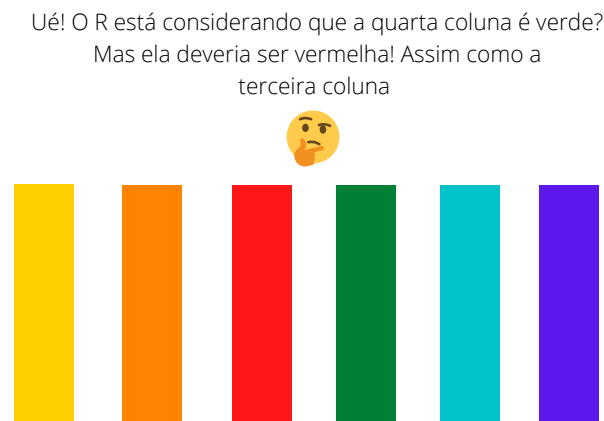
Fonte: Elaboração própria do autor.

Além da forma como os nossos dados estão organizados dentro do R, nós podemos estar interessados também na forma em que o R está interpretando os nossos dados, em um dado momento. Neste caso, estamos nos perguntando qual o tipo de dado que o R está associando aqueles valores, e em

muitas ocasiões podemos nos surpreender com as escolhas da linguagem. Uma surpresa, que está representada na figura 2.5. Em resumo, quando eu vejo o valor "20/05/2020", eu rapidamente o associo à data 20 de maio de 2020, mas será que o R compreende que este valor se trata de uma data?

Pelo fato das datas não estarem entre os tipos de dados básicos do R, ele não vai identificar sozinho que aquele valor se trata de uma data, até que a gente diga isso a ele. Até lá, o R irá interpretar este valor como um simples texto. Isso é um ponto importante, pois várias funções ou ações que queremos executar no R, exigem que os seus dados estejam no tipo adequado. Por isso, você vai enfrentar diversas situações onde o console lhe retorna um erro confuso, e depois de alguns minutos, você busca conferir a estrutura de seus dados, e descobre que o R estava o tempo todo interpretando os seus números como textos!

Figura 2.5: Tipos de dados



Fonte: Elaboração própria do autor.

Portanto, vamos começar descrevendo nas próximas seções as estruturas de dados presentes na linguagem, e em seguida, os tipos de dados básicos do R. Até onde me recordo, tem apenas uma estrutura do R em específico, que não vou descrever nas próximas seções, que é o array. Nós veremos mais a frente, as matrizes, que no R são vetores com duas dimensões (uma dimensão para as linhas e outra para as colunas). O array também é (assim como a matriz) um vetor com mais de uma dimensão, porém, ele pode ser um vetor com “n” dimensões. Em outras palavras, com um array você pode criar um objeto tridimensional (3 dimensões), ou se quiser ir longe, um objeto com 4, 5, ou infinitas dimensões.

2.4 Estruturas de dados

2.4.1 Vetores

Os vetores são a estrutura básica da linguagem R, pois todas as outras estruturas, são construídas a partir desses vetores. Um vetor é simplesmente uma sequência de valores. Valores que podem ser datas, números, textos, índices, ou qualquer outro tipo que você imaginar. Pelo fato de ser uma simples sequência de valores, o vetor é uma estrutura unidimensional. É como se esse vetor fosse composto por apenas uma coluna, que você preenche com quantas linhas você precisar. Ou então, você também pode imaginá-lo como uma corda, que amarra e mantém os seus valores conectados um atrás do outro.

A forma mais simples de se criar um vetor, é através da função `c()` (abreviação para *combine*, ou combinar), em que você fornece os valores que quer incluir neste vetor, separando-os por vírgulas. A outra forma (indireta) de se criar um vetor, é através de funções que retornam por padrão este tipo de estrutura. Um exemplo simples, é a função `1:10` : que serve para criar sequências numéricas no R, no exemplo abaixo, uso essa função para criar uma sequência de 1 a 10. Outro exemplo, seria a função `rep()` que serve para repetir um conjunto de valores, por quantas vezes você quiser.

```
c(48, 24, 12, 6)
```

```
## [1] 48 24 12 6
```

```
c("a", "b", "c", "d")
```

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

```
1:10
```

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

```
rep(c("Ana", "Eduardo"), times = 5)
```

```
## [1] "Ana"      "Eduardo" "Ana"      "Eduardo" "Ana"      "Eduardo" "Ana"
## [8] "Eduardo" "Ana"      "Eduardo"
```

Como o vetor é uma estrutura unidimensional, eu posso acessar um único valor dentro desse vetor, utilizando apenas um índice. Por exemplo, se eu quero extrair o quarto valor dessa sequência, eu utilizo o número 4, se eu quero o terceiro valor, o número 3, e assim por diante. Para acessar “partes”, ou um único valor de uma estrutura no R, nós utilizamos a função `[]`, e para utilizá-la, basta abrir colchetes após o nome do objeto onde você salvou este vetor, ou após a função que está gerando este vetor.

```
vetor <- 1:10
```

```
vetor[4]
```

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

```
c("a", "b", "c")[3]
```

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

Para acessar mais de um valor dentro deste vetor, você terá que fornecer um novo vetor de índices à função `[]`. Um jeito prático de criar este novo vetor de índices, é criando uma sequência com a função `:` que vimos anteriormente. Um detalhe, é que o R irá extrair os valores na ordem em que você os dá a `[]`. Logo, se eu dentro de `[]` incluir o vetor `c(2, 4, 6, 1)`, o R irá lhe retornar um novo vetor, que contém o segundo, quarto, sexto e primeiro item do vetor anterior, respectivamente. Caso você repita algum índice, o R irá repetir o valor dentro do vetor resultante, e não te avisará sobre isso.

```
vetor <- 1:25
```

```
vetor[1:4]
```

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

```
vetor[8:13]
```

```
## [1] 8 9 10 11 12 13
```

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

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

Os vetores que estamos criando com essas funções são comumente chamados de vetores atômicos (*atomic vector*). Esses vetores possuem uma propriedade simples e importante: **vetores atômicos possuem apenas um único tipo de dado dentro deles**. Você não consegue guardar dentro de um mesmo vetor, valores de dois tipos de dados diferentes (por exemplo, textos e números) sem que alguma transformação ocorra. Caso você tente burlar essa regra, o R irá automaticamente converter os valores para um único tipo de dado, e pode ser que parte desses dados não possam ser convertidos de forma lógica para este único tipo, e acabam sendo “perdidos” neste processo. Falaremos mais sobre esse processo de conversão, quando chegarmos em tipos de dados.

2.4.2 Matrizes

Matrizes nada mais são do que vetores com duas dimensões. Se você possui dados atualmente alocados em um vetor, e deseja organizá-los em colunas e linhas, você pode rapidamente criar uma matriz com este vetor, ao adicionar dimensões a ele, através da função `dim()`. Você usa a função sobre o vetor desejado à esquerda do símbolo de *assignment* (`<-`), e atribui um valor ao resultado dessa função. No caso de matrizes, esse valor será um vetor com dois elementos, o primeiro definindo o número de linhas, e o segundo, o número de colunas.

```
vetor <- 1:6

dim(vetor) <- c(3,2)

vetor
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Uma outra forma de criar uma matriz, é através da função `matrix()`. Você primeiro fornece um vetor à função, e define quantas colunas você deseja em `ncol`, e quantas linhas em `nrow`. Um detalhe que fica claro no exemplo abaixo, é que ao criar uma matriz, ela por padrão será preenchida por coluna, e não por linha. Caso você queira que ela seja preenchida por linha, você deve adicionar o valor `TRUE`, ao argumento `byrow` na função.

```
# Para preencher a matriz, por linha, adicione
# byrow = TRUE à função
matrix(1:20, nrow = 5, ncol = 4)

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

Os vetores são estruturas unidimensionais, e com apenas um índice poderíamos acessar um valor contido nele. Porém, as matrizes possuem duas dimensões, logo, teremos que fornecer dois índices à função `[` para acessarmos um único elemento dessa matriz. Basta você separar esses dois índices por uma vírgula, onde o primeiro valor corresponde a linha, e o segundo, a coluna desejada. No exemplo abaixo, estou extraindo o elemento que se encontra na terceira linha da quarta coluna.

```
matriz <- matrix(1:20, nrow = 5, ncol = 4)

matriz[3,4]
```

```
## [1] 18
```

Eu posso também extrair uma parte dessa matriz, ao fornecer mais valores dentro de um vetor, para cada um dos dois índices. No primeiro exemplo abaixo, eu extraio todos os valores da primeira a terceira linha da segunda coluna da matriz. Agora, caso eu queira extrair todos os valores de uma dimensão (todas as linhas, ou todas as colunas), basta que eu deixe em “branco” o lado de cada índice. No segundo exemplo abaixo, estou extraindo todos os valores da segunda coluna.

```
matriz[1:3, 2] # É o mesmo que: matriz[c(1,2,3), 2]
```

```
## [1] 6 7 8
```

```
matriz[, 2]
```

```
## [1] 6 7 8 9 10
```

Pelo fato de matrizes serem vetores com duas dimensões, elas herdam a propriedade do vetor, e portanto: **matrizes podem conter dados de apenas um único tipo**. Por essa característica, você provavelmente utilizará essa estrutura poucas vezes. De qualquer forma é útil conhecê-la.

2.4.3 Listas

A lista é uma estrutura especial e muito importante do R, pois ela é a exceção da propriedade dos vetores (que podem conter apenas um tipo de dado). **Portanto, uma lista é um vetor, onde cada elemento deste vetor pode ser não apenas de um tipo de dado diferente, mas também de tamanho e estrutura diferentes**. Dito de outra forma, você pode incluir o que você quiser em cada elemento de uma lista.

Uma lista é criada pela função `list()`, e para utilizá-la, basta fornecer os valores que deseja inserir em cada elemento desta lista, separados por vírgulas. No exemplo abaixo, estou inserindo no primeiro elemento desta lista a data que vimos anteriormente (“20/05/2020”), no segundo, estou incluindo uma matriz, no terceiro, um vetor com nomes, e no quarto, um `data.frame` (falaremos sobre eles após essa seção).

```
# Lista nomeada
# nome = valor
lista <- list(
  data = "20/05/2020",
  matriz = matrix(1:20, ncol = 4, nrow = 5),
  vetor = c("Belo Horizonte", "Londrina", "Macapá"),
  tabela = data.frame(x = 21:30, y = rnorm(10))
)

lista
```

```
## $data
## [1] "20/05/2020"
##
## $matriz
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
##
## $vetor
## [1] "Belo Horizonte" "Londrina"      "Macapá"
##
## $tabela
##      x      y
## 1  21  1.3742702
## 2  22 -0.3201668
## 3  23  0.5524656
## 4  24 -0.6353341
## 5  25 -1.5974158
## 6  26  1.0616887
## 7  27  0.3870726
## 8  28  0.8618662
## 9  29  0.2313124
## 10 30 -0.8338460
```

Perceba que nós nomeamos cada elemento dessa lista. Isso abre novas possibilidades, pois agora podemos utilizar um sistema diferente da função `[]` para acessarmos os valores específicos de uma lista, utilizando o operador `$`. Através deste operador, podemos acessar os elementos dessa lista, através do nome que demos para cada um deles. O problema deste sistema, é que ele lhe permite acessar todos os valores contidos em um elemento de sua lista, mas não lhe permite extrair valores específicos contidos em cada um destes elementos da lista.

```
lista$matriz
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
lista$vetor
```

```
## [1] "Belo Horizonte" "Londrina" "Macapá"
```

Você não precisa nomear cada um dos elementos dessa lista como fizemos acima. Eu nomeie apenas para dar um exemplo do operador \$. Porém, neste caso em que você não atribui um nome a esses elementos, você não pode acessá-los mais pelo operador \$, e terá que retornar à função [para tal serviço. Em outras palavras, se você deseja criar uma lista, mas não está muito preocupado em nomear cada um dos elementos que vão estar nessa lista, basta separar esses valores por vírgulas como no exemplo abaixo:

```
lista <- list(
  c(6, 7, 8),
  c("a", "b", "c"),
  c(T, F, T)
)
```

```
lista
```

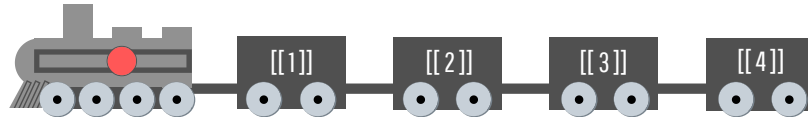
```
## [[1]]
## [1] 6 7 8
##
## [[2]]
## [1] "a" "b" "c"
##
## [[3]]
## [1] TRUE FALSE TRUE
```

Antes de prosseguirmos, darei uma nova descrição (dessa vez, uma descrição visual) de uma lista, para que você fixe na sua cabeça o que ela é. Eu espero que eu tenha desejado bem o suficiente, para que você seja capaz de identificar um trem carregando quatro vagões na figura 2.6. Podemos pensar esse trem como uma lista, e os seus vagões como os elementos dessa lista. Tendo isso em mente, temos na figura 2.6 uma representação de uma lista com quatro elementos.

Como disse anteriormente, podemos incluir o que quisermos dentro de cada elemento dessa lista, ou dentro de cada vagão desse trem. Pois cada vagão é capaz de comportar elementos de qualquer dimensão e em qualquer estrutura, e como esses vagões estão separados uns dos outros, esses elementos não precisam compartilhar das mesmas características. Dito de outra forma, eu posso carregar 15 toneladas de ouro no primeiro vagão, 100 Kg de carvão no segundo vagão, e 1 Kg de ferro no terceiro vagão.

Portanto, a lista é uma estrutura que lhe permite transportar todos esses diferentes elementos, em um mesmo objeto no R (ou todos esses diferentes componentes em um mesmo trem). Quando chegarmos em interação, você verá que essa característica torna a lista, uma estrutura extremamente útil.

Agora como eu posso extrair valores dessa lista através da função [? Bem, a lista é a exceção da propriedade dos vetores, mas ela continua sendo um vetor em sua essência, ou uma estrutura

Figura 2.6: Representação de uma lista

Fonte: Elaboração própria do autor.

unidimensional. Por isso, você pode acessar um item de uma lista com apenas um índice dentro de `[`.

Porém, caso você usar apenas um colchete para selecionar o primeiro elemento de sua lista, você percebe que uma pequena descrição ("`[[1]]`"), ou o nome que você deu aquele elemento, aparece em cima dos valores contidos neste elemento da lista. Por isso, se você deseja extrair apenas os valores desse elemento, sem essa descrição, você deve utilizar o índice dentro de dois colchetes.

```
lista <- list(
  1:20,
  "0 ano tem 365 dias",
  matrix(1:20, ncol = 4, nrow = 5)
)

lista[1]

## [[1]]
## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

lista[[1]]

## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

lista[[2]]

## [1] "0 ano tem 365 dias"

lista[[3]]

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
```

```
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

Isso ocorre, porque quando você utiliza apenas um colchete para selecionar o primeiro elemento, o R acaba lhe retornando uma nova lista contendo um elemento, e não apenas o que está dentro deste elemento em si. Dizendo em termos da representação visual que utilizamos na figura 2.6, se eu possuo um trem com quatro vagões, e utilizo um colchete para selecionar o primeiro vagão, o R me retorna um novo trem que contém o primeiro vagão. Mas se eu utilizo dois colchetes, o R me retorna apenas o primeiro vagão, e nada mais.

Figura 2.7: Diferença entre um e dois colchetes em listas



Fonte: Elaboração própria do autor.

Mas como eu faço para extrair um valor específico de um elemento de uma lista? Para isso você deve abrir um novo colchete após os colchetes duplos que você criou para selecionar o elemento da lista. A partir daí, basta replicar o que vimos anteriormente com os índices. No exemplo abaixo, estou primeiro selecionando o terceiro elemento da nossa lista (que é uma matriz), e selecionando o item da terceira linha da primeira coluna desta matriz.

```
lista[[3]][3,1]
```

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

2.4.4 Tabelas no R: data.frame

O data.frame é a principal estrutura utilizada para guardar tabelas e bases de dados no R. Na grande maioria das vezes que você importar os seus dados para o R, eles serão alocados dentro de um data.frame. Essa estrutura é no fundo, uma lista com algumas propriedades a mais. Por isso, o data.frame herda uma de suas principais propriedades: **cada uma das colunas da tabela formada por um data.frame, pode conter um tipo de dado diferente das demais colunas deste data.frame.**

Esta é uma das principais características que tornam o data.frame, uma estrutura adequada para guardar a grande maioria das bases de dados. Pois é muito comum, que você possua em sua base,

diversas colunas contendo dados de diferentes tipos. Por exemplo, você pode ter uma base que possui uma coluna contendo datas, outras duas contendo valores numéricos, e uma última coluna contendo textos, ou rótulos indicando a qual indicador ou grupo, os valores numéricos da linha se referem. E ao importar uma base como essa para o R, é de seu desejo que o R interprete essas colunas corretamente e mantenha os tipos desses dados intactos.

Os `data.frame`'s são criados pela função `data.frame()`. Você deve preencher essa função com os valores que você deseja alocar em cada coluna separados por vírgulas. Você pode escolher não dar um nome a cada coluna, neste caso a função se ocupará de dar um nome genérico para elas. Caso opte por definir esses nomes, você deve fornecê-los antes dos valores da coluna, seguindo a seguinte estrutura:

```
# Estrutura Básica:
# data.frame(
#   <nome_coluna> = <valor_coluna>
# )

data.frame(
  nomes = rep(c("Ana", "Eduardo"), times = 5),
  numeros = rnorm(10),
  constante = 25
)
```

##	nomes	numeros	constante
## 1	Ana	2.0623664	25
## 2	Eduardo	-1.2145019	25
## 3	Ana	1.3763474	25
## 4	Eduardo	0.7903205	25
## 5	Ana	-0.8359929	25
## 6	Eduardo	0.0713462	25
## 7	Ana	0.5397631	25
## 8	Eduardo	2.3611914	25
## 9	Ana	-0.1520724	25
## 10	Eduardo	-0.2603465	25

Caso você esteja em dúvida, tudo o que a função `rnorm()` faz é gerar valores aleatórios seguindo uma distribuição normal. Vemos que no exemplo acima, geramos uma tabela com 3 colunas e 10 linhas, e aqui chego a segunda principal propriedade de um `data.frame`, que é: **todas as colunas de um `data.frame` devem possuir o mesmo número de linhas**. O motivo dessa propriedade é um pouco óbvio, pois se estamos tentando formar uma tabela de dados, é natural pensarmos que ela deve formar um retângulo uniforme.

Isso significa, que se eu pedisse para a função `rep()` repetir os valores 6 vezes (ao invés de 5), gerando assim um vetor de 12 elementos (ou 12 linhas), a função `data.frame()` me retornaria um erro, indicando que o número de linhas criadas pelos diferentes vetores não possuem o mesmo número de linhas.

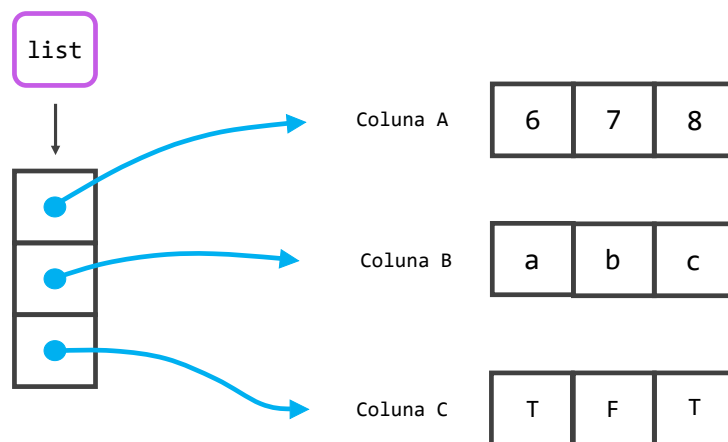
Caso não tivéssemos essa propriedade, estaríamos permitindo que alguma dessas colunas deste `data.frame`, fosse mais longa do que as outras. Neste caso, como você lidaria com as observações “sobressalentes” da tabela? Você possui um valor na coluna x que não possui um valor correspondente na coluna y, será que você considera o valor da coluna y como vazio? Não disponível? Não existente? Enfim, uma confusão que é desnecessária.

Essa propriedade nos garante que para cada observação (ou linha) da nossa tabela, deve **sempre** existir um valor na coluna y correspondente ao valor da coluna x, mesmo que o valor da coluna y seja um valor NA (não disponível), ou algo indicando que não foi possível coletar esse valor no plano físico de nossa atividade.

Ao voltar para o exemplo acima, você pode perceber que na terceira coluna que definimos em `data.frame()`, demos uma simples constante (25) à função. Como resultado, a função acaba preenchendo toda a coluna por essa constante. Isso ocorre sempre que você fornece um único valor a uma coluna de seu `data.frame`, seja este valor, uma data, um texto, um número ou qualquer outro tipo que imaginar.

A partir daqui, é interessante criarmos um modelo visual em nossa cabeça, sobre o que um `data.frame` representa. Como disse anteriormente, um `data.frame`, é basicamente uma lista, com algumas propriedades a mais, em especial a propriedade de que todos os seus elementos devem possuir o mesmo número de linhas. Portanto, se você quer imaginar um `data.frame` em sua mente, você pode imaginar uma lista, onde cada um de seus elementos, representa uma coluna desse `data.frame`. Em conjunto, essas colunas (ou os elementos dessa lista) formam uma tabela, sendo essa tabela, comumente referida como um `data.frame`.

Figura 2.8: Representação de um `data.frame` a partir de uma lista



Fonte: Elaboração própria do autor.

Vale destacar um outro comportamento da função `data.frame()`. Ela transforma por padrão, todos os textos em fatores (*factor*), ou em outras palavras, valores de uma variável categórica que possui um conjunto limitado de valores possíveis. Vamos aprender mais sobre este tipo de dados nas próximas seções. Inicialmente, isso não tem grandes implicações sobre os seus dados. Eles vão continuar sendo apresentados como textos, e a única grande mudança será sobre a forma como o R irá ordenar esses valores caso você peça isso a ele. Mas é importante saber deste detalhe, pois você vai querer suprimir esse comportamento na maioria das vezes. Para isso, basta adicionar o valor `FALSE` para o argumento `stringsAsFactors`.

```
tabela <- data.frame(
  cidade = rep(c("Belo Horizonte", "Londrina", "Macapá"), times = 4),
  valor = rnorm(12),
  stringsAsFactors = FALSE
)

# Estou utilizando a função is.character()
# para confirmar que data.frame() manteve
# a coluna de cidades como texto (characters)
is.character(tabela$cidade)

## [1] TRUE
```

No exemplo acima, você também percebe que eu utilizei dentro da função `is.character()`, o operador `$` para acessar os valores da coluna `cidade` da nossa tabela. Em `data.frame`'s você sempre pode utilizar este mecanismo para acessar os valores de uma das colunas de sua tabela, pois `data.frame()` irá sempre se preocupar em nomear as colunas caso você não o faça. Portanto, mesmo que `data.frame()` invente um nome completamente esquisito para as suas colunas, elas sempre terão um nome para o qual você pode se referir com `$`.

Isso não significa que você deixará de utilizar o sistema `[]`, pois essa função é muito mais flexível do que você imagina. Uma de suas principais e mais poderosas ferramentas, é um sistema que é comumente chamado de *logical subsetting*. Com ele, podemos usar a função `[]` para extrair valores de um objeto, de acordo com o resultado de testes lógicos. Em diversas funções de pacotes que você utilizar, se você visitar o código fonte dessas funções, você irá encontrar este sistema sendo utilizado em algum momento, sendo portanto, uma ferramenta extremamente útil dentro do R.

Em resumo, se você quer extrair todos os valores de uma coluna de seu `data.frame`, você pode utilizar o sistema `$`, ou o mesmo sistema que utilizamos em matrizes, ao deixar o índice das linhas em “branco” dentro de `[]`. Se você quer extrair partes específicas de sua tabela, você terá que usar `[]` da mesma forma que o utilizamos em matrizes. Como as colunas de um `data.frame` são nomeados, você pode também extrair uma coluna inteira, ao colocar o nome dessa coluna entre aspas dentro dos colchetes. Todos os sistemas utilizados abaixo, nos retorna todos os valores da coluna `cidade`.

```
tabela$cidade
```

```
tabela[, 1]
```

```
tabela[["cidade"]]
```

Você deve ter percebido acima que utilizei novamente os dois colchetes, ao me referir dentro deles pelo nome da coluna desejada. Este sistema funciona exatamente da mesma forma que ele funciona em listas. Se eu utilizar um colchete, o R me retorna um `data.frame` contendo uma única coluna (neste caso, a coluna `cidade`), se eu uso dois colchetes, o R me retorna um vetor contendo apenas os valores dessa coluna.

Agora, voltando um pouco em nossa descrição, quando eu disse que um `data.frame` são listas, pois herdava muitas de suas propriedades, eu acabei omitindo uma dessas propriedades para evitar confusões. Você deve ter percebido pelos exemplos anteriores, que cada elemento de um `data.frame` é uma coluna de sua tabela. Você talvez tenha percebido também que todos esses elementos nos exemplos anteriores, eram vetores. Isso é uma característica marcante de um `data.frame`, pois na maioria das vezes em que você ver um, ele estará servindo apenas como um laço, que amarra e mantém diferentes vetores unidos em uma mesma estrutura, vetores esses que juntos formam uma tabela.

Você deve estar pensando: “Mas é claro que cada coluna é um vetor! Não faria sentido se eu incluísse matrizes ou outras tabelas em uma coluna de uma tabela! Um vetor é a estrutura que faz mais sentido para essas colunas!”. Bom, eu creio que agora é uma boa hora para “explodir” a sua cabeça!...ou pelo menos metaforicamente falando. **A outra propriedade que `data.frame`'s herdam de listas, é que cada um de seus elementos também não precisam ser da mesma estrutura.**

Essa propriedade significa que eu posso incluir sim, uma matriz, ou um outro `data.frame`, como uma nova coluna de um `data.frame` que está salvo em algum objeto. Lembre-se que a principal diferença entre um `data.frame` e uma lista, é que os elementos de um `data.frame` precisam obrigatoriamente ter o mesmo número de linhas. No exemplo abaixo, eu estou criando inicialmente um `data.frame` com 10 linhas e 2 colunas, logo, se eu quiser incluir uma nova tabela como uma nova coluna desse `data.frame`, essa nova tabela (ou novo `data.frame`) deve possuir 10 linhas (mas esse novo `data.frame` pode ter quantas colunas você desejar).

Você pode facilmente adicionar uma nova coluna a um `data.frame`, utilizando o operador `$`. Você escreve primeiro o nome do objeto onde o seu `data.frame` está contido, abre o cifrão (`$`), e em seguida, coloca um nome de uma coluna que não existe em seu `data.frame` até aquele momento. Se não há alguma coluna neste `data.frame` que possui este nome, o R irá adicionar esta coluna a ele, e para você preencher essa coluna com algum valor, basta utilizar o símbolo de *assignment* (`<-`), como se você estivesse salvando algum valor em um novo objeto. Após criar essa nova coluna, eu chamo por ela, para que o R me mostre o que tem nessa coluna, e como esperávamos, ele me retorna o novo `data.frame` que criamos.

```
tabela <- data.frame(  
  cidade = rep(c("Belo Horizonte", "Londrina"), times = 5),  
  valor = rnorm(10)
```

```
)

tabela$novovo_dataframe <- data.frame(
  x = rep("Ana", times = 10),
  y = rep("Eduardo", times = 10),
  z = 25
)

tabela$novovo_dataframe
```

```
##      x      y  z
## 1 Ana Eduardo 25
## 2 Ana Eduardo 25
## 3 Ana Eduardo 25
## 4 Ana Eduardo 25
## 5 Ana Eduardo 25
## 6 Ana Eduardo 25
## 7 Ana Eduardo 25
## 8 Ana Eduardo 25
## 9 Ana Eduardo 25
## 10 Ana Eduardo 25
```

Na figura 2.5, estou utilizando a função `str()` sobre o objeto `tabela`. Essa função nos retorna no console, uma descrição da estrutura de um objeto. No retângulo vermelho, temos a estrutura geral do objeto, vemos que o objeto `tabela` é um `data.frame` com dez linhas e três colunas. Os nomes de suas três colunas estão especificadas no retângulo verde. A direita do nome da terceira coluna (chamada `novovo_dataframe`), podemos ver uma descrição de sua estrutura marcada por um retângulo azul. Vemos neste retângulo azul, portanto, a estrutura desta terceira coluna, e podemos confirmar que se trata também de um `data.frame` com 10 linhas e 3 colunas, e no retângulo roxo, podemos ver o nome das três colunas (no caso abaixo, colunas `x`, `y` e `z`) contidas neste segundo `data.frame`. Os falantes de língua inglesa costumam se referir a esta situação onde inserimos uma nova estrutura dentro de uma mesma estrutura, como uma *nested structure*, ou uma estrutura “aninhada”. Logo, o exemplo que estou dando, se trata de um *nested data.frame*. Pois estamos inserindo um `data.frame`, dentro de um outro `data.frame`.

Se você chamar pelo nome `tabela` no console, para ver o que tem dentro deste objeto, o console irá lhe mostrar um `data.frame` com 10 linhas e 5 colunas. Pois ele lhe apresenta tanto as 2 colunas definidas como vetores em `tabela`, quanto as 3 colunas definidas em `tabela$novovo_dataframe`, tudo em uma mesma tabela. Entretanto, como vimos através da função `str()`, o R está considerando este objeto como um `data.frame` com 10 linhas e 3 colunas, onde a terceira coluna contém um novo `data.frame` de 10 linhas e com outras 3 colunas, e não como um único `data.frame` com 10 linhas e 5 colunas.

Tendo essas considerações em mente, você pode sim incluir dados que estão em qualquer uma das estruturas anteriormente mencionadas, dentro de uma coluna (ou elemento) de um `data.frame`.

Figura 2.9: Estrutura de um data.frame aninhado

```

> tabela$novos_dataframe <- data.frame(
+   x = rep("Ana", times = 10),
+   y = rep("Eduardo", times = 10),
+   z = 25
+ )
> str(tabela)
'data.frame': 10 obs. of 3 variables:
 $ cidade      : chr  "Belo Horizonte" "Londrina" "Belo Horizonte" "Londrina" ...
 $ valor       : num  -0.299 -1.313 -0.283 -0.483 -0.968 ...
 $ novos_dataframe: data.frame': 10 obs. of 3 variables:
 .. $ x: chr  "Ana" "Ana" "Ana" "Ana" ...
 .. $ y: chr  "Eduardo" "Eduardo" "Eduardo" "Eduardo" ...
 .. $ z: num  25 25 25 25 25 25 25 25 25 25

```

Fonte: Elaboração própria do autor.

Essa propriedade é mais citada nos manuais originais da linguagem (R CORE TEAM, 2020b; R CORE TEAM, 2020a), enquanto é muito pouco mencionada, ou pouco explicada em detalhes em outros livros-texto sobre a linguagem. Pois é uma propriedade que faz pouco sentido, considerando-se as principais aplicações de um data.frame. Porém, com essa propriedade, você pode pensar facilmente em uma outra estrutura que é muito mais útil e muito mais poderosa, para ser incluída em uma nova coluna de seu data.frame. Essa estrutura, é uma lista!

Pense um pouco sobre isso. Uma lista é um vetor em sua essência, e por isso, pode facilmente formar uma nova coluna desse data.frame. A vantagem de se incluir uma lista, é que agora em cada célula (ou em cada linha) dessa nova coluna, eu posso guardar um dado de um tipo, tamanho e estrutura diferentes. Se fossemos utilizar a representação visual da seção anterior, é como se a coluna de seu data.frame tenha se transformado em um trem, e agora cada célula, ou cada linha dessa coluna, tenha se tornado um vagão deste trem. Com essa realidade, você pode por exemplo, facilmente aplicar um modelo de regressão sobre 1.000 bases de dados diferentes, e ainda guardar os resultados em cada linha de uma nova coluna, tudo isso com apenas um comando! Dessa forma, você terá em uma coluna de seu data.frame contendo uma lista, lista essa que está mantendo todos esses 1.000 data.frame's diferentes juntos.

Se você consegue entender a língua inglesa, mesmo que sutilmente, eu altamente recomendo que assista a palestra de Hadley Wickham, intitulada “*Managing many models with R*”, que está disponível no YouTube¹. Nesta palestra, ele dá um exemplo prático de como você pode implementar essa ideia, ao aplicar um modelo de regressão sobre várias bases diferentes, utilizando essa propriedade em um data.frame.

¹https://www.youtube.com/watch?v=rz3_FDvt9eg

2.4.5 tibble's como uma alternativa moderna aos data.frame's

Um tibble nada mais é do que uma “versão moderna” de um `data.frame`. Essa estrutura de dado é originária do pacote `tibble`, logo, se você deseja utilizá-la em algum de seus dados, você terá que chamar obrigatoriamente por esse pacote com o comando `library()`². Lembre-se que o pacote deve estar instalado em sua máquina, para que você seja capaz de chamar por ele com o comando `library()`.

Portanto, essa estrutura foi criada com o intuito de melhorar alguns comportamentos do `data.frame`, que eram adequados para a sua época, mas que hoje, são desnecessários e que podem gerar um pouco de dor de cabeça. Tais estruturas podem ser criadas do zero, através da função `tibble()`, que funciona da mesma maneira que `data.frame()`. Você dá o nome para cada coluna, e após um igual (=) você define o que irá preencher cada uma dessas colunas.

```
library(tibble)
```

```
tab_tibble <- tibble(
  Datas = seq.Date(as.Date("2020-12-01"), as.Date("2020-12-10"), by = 1),
  Usuario = sample(c("Ana", "Eduardo"), size = 10, replace = T),
  Valor = sample(c(2000, 3000, 4000, 5000), size = 10, replace = T)
)
```

```
tab_tibble
```

```
## # A tibble: 10 x 3
##   Datas      Usuario Valor
##   <date>    <chr>   <dbl>
## 1 2020-12-01 Eduardo  5000
## 2 2020-12-02 Ana      2000
## 3 2020-12-03 Eduardo  2000
## 4 2020-12-04 Eduardo  3000
## 5 2020-12-05 Ana      3000
## 6 2020-12-06 Ana      5000
## 7 2020-12-07 Ana      4000
## 8 2020-12-08 Eduardo  2000
## 9 2020-12-09 Ana      3000
## 10 2020-12-10 Ana      5000
```

Por outro lado, se você já possui um `data.frame` e deseja convertê-lo para um tibble, você precisa apenas aplicar a função `as_tibble()` sobre ele.

```
tabela <- as_tibble(tabela)
```

²Caso tenha alguma dificuldade em chamar pelo pacote, volte a seção [Pacotes](#) para descobrir o passo que você se esqueceu de cumprir.

A primeira melhoria dessas estruturas, se encontra no método de `print()`, ou em outras palavras, na forma como o R lhe mostra a sua tabela no console. Quando chamamos por um objeto que é um `data.frame`, o console acaba lhe retornando muito mais linhas do que o necessário (ele pode retornar até 1000 linhas), além de todas as colunas da tabela. Se o seu `data.frame` possui várias colunas, você pode se sentir frustrado com esse comportamento, pois se alguma coluna de sua tabela não couber ao lado das colunas anteriores, o console acaba quebrando o resultado em várias “linhas”, algo que pode tornar a leitura confusa com certa facilidade.

As origens do R são antigas (> 50 anos), e aparentemente esse não era um comportamento muito ruim na época, talvez porque as dimensões das tabelas dessa época eram muito limitadas. Porém, com as capacidades de processamento atuais, essa atitude é desnecessária ou indesejada em quase todas as situações. Veja no exemplo abaixo, onde eu pego a base `flights` (que possui 19 variáveis diferentes), e transformo-a em um `data.frame` com a função `as.data.frame()`. Para que o resultado não consuma muito espaço deste material, eu ainda limito o resultado às 5 primeiras linhas da tabela com `head()`. Perceba que a tabela foi dividida em 3 linhas diferentes de *output*.

```
library(nycflights13)
```

```
as.data.frame(flights) %>%
  head(n = 5)
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 1 2013     1   1     517           515         2      830           819
## 2 2013     1   1     533           529         4      850           830
## 3 2013     1   1     542           540         2      923           850
## 4 2013     1   1     544           545        -1     1004          1022
## 5 2013     1   1     554           600        -6      812           837
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1         11      UA   1545  N14228   EWR  IAH      227      1400    5
## 2         20      UA   1714  N24211   LGA  IAH      227      1416    5
## 3         33      AA   1141  N619AA   JFK  MIA      160      1089    5
## 4        -18      B6    725  N804JB   JFK  BQN      183      1576    5
## 5        -25      DL    461  N668DN   LGA  ATL      116       762    6
##   minute          time_hour
## 1      15 2013-01-01 05:00:00
## 2      29 2013-01-01 05:00:00
## 3      40 2013-01-01 05:00:00
## 4      45 2013-01-01 05:00:00
## 5       0 2013-01-01 06:00:00
```

Quando as suas tabelas são `tibble`'s, o console lhe retorna por padrão, apenas as 10 primeiras linhas da tabela (caso a tabela seja muito pequena, ele pode lhe retornar todas as linhas), o que já é o suficiente para vermos a sua estrutura. Além disso, caso as próximas colunas não caibam em uma mesma “linha”, ou ao lado das colunas anteriores, o `tibble` acaba omitindo essas colunas para não

sobrecarregar o seu console de resultados. Lembre-se que você sempre pode ver toda a tabela, em uma janela separada através da função `View()`.

```
View(flights)
```

Veja o exemplo abaixo, onde eu chamo novamente pela base `flights`. O primeiro detalhe que você percebe, é a dimensão da tabela (algo que não é informado, quando chamamos por um `data.frame`) no canto superior esquerdo da tabela (336.776 linhas e 19 colunas). O segundo detalhe, é que o tipo de dado contido em cada coluna, está descrito logo abaixo do nome da coluna, de acordo com a abreviação deste tipo. Por exemplo, nas três primeiras colunas estão contidos números inteiros (*integer's* - `int`), enquanto na sexta coluna (`dep_delay`) temos números decimais (*double's* - `dbl`).

Mesmo que em um `tibble`, você fique sem a possibilidade de visualizar todas as outras colunas da tabela, que não cabem na mesma linha junto com as colunas anteriores, um `tibble` sempre lhe retorna logo abaixo da tabela, uma lista contendo o nome de todas as colunas restantes, além do tipo de dado contido em cada coluna, através das mesmas abreviações que vimos nas colunas anteriores.

```
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>
## 1  2013     1     1     517             515           2     830
## 2  2013     1     1     533             529           4     850
## 3  2013     1     1     542             540           2     923
## 4  2013     1     1     544             545          -1    1004
## 5  2013     1     1     554             600          -6     812
## 6  2013     1     1     554             558          -4     740
## 7  2013     1     1     555             600          -5     913
## 8  2013     1     1     557             600          -3     709
## 9  2013     1     1     557             600          -3     838
## 10 2013     1     1     558             600          -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Além desses pontos, `tibble's` vão sempre criar destaques, ou ênfases em certos dados no console, algo que os `data.frame's` não fazem em nenhum momento. Por exemplo, `tibble's` vão sempre marcar de vermelho, qualquer número que seja negativo, uma funcionalidade que é bem familiar aos usuários de Excel que utilizam formatação condicional. Um outro detalhe, é que essa estrutura também marca as casas dos milhares com um pequeno sublinhado, o que facilita muito a leitura de números muito grandes.

Para mais, um comportamento muito comum de um `data.frame`, é converter os seus dados em textos, para fatores (`factor`). Este não é um comportamento de todo ruim, e nem sempre ele ocorre.

Figura 2.10: Ênfase em valores numéricos presentes em um tibble

```
> tibble(x = valores)
# A tibble: 7 x 1
      x
  <dbl>
1  8920
2 -2290
3 20100
4 310040
5 -12500
6  1635
7  7321
```

Fonte: Elaboração própria do autor.

Porém o principal valor dos fatores no R, está no uso de *dummies* em regressões e análises estatísticas, além da maneira como a ordenação de seus valores é executada. Estas características são importantes, mas também são irrelevantes para uma gama muito grande de situações. Em outras palavras, este é um comportamento desnecessário na maioria de nossas análises.

Por isso, uma outra característica que os tibble's carregam, é que eles nunca transformam os seus dados para um outro tipo. Isso é um ponto muito importante! As funções com as quais nós trabalhamos no R, geralmente funcionam melhor com (ou são especializadas em) uma estrutura ou tipo de dado específico, e quando nós estruturamos as nossas análises sobre essas funções, nós desejamos evitar mudanças não autorizadas sobre os tipos e estruturas utilizados.

Ou seja, é sempre melhor evitar transformações implícitas de seus dados. Pois essas operações podem muito bem, levantar erros dos quais você não compreende, até que você (depois de muito tempo analisando os resultados) perceba que os seus dados foram convertidos para algo incompatível com o que você deseja realizar.

Dessa forma, em um tibble os seus dados em texto são interpretados como textos (character), a menos que você peça explicitamente ao R que interprete esses dados de uma outra forma. Veja o exemplo abaixo, onde utilizo a função `str()` para ver um resumo da estrutura de cada tabela. Podemos ver abaixo, que a coluna `text` na tabela `tib` contém dados do tipo `character` (`chr`), enquanto essa mesma coluna na tabela `df`, possui dados do tipo `factor`.

```
tib <- tibble(
  x = rnorm(10),
  text = sample(c("Ana", "Eduardo"), size = 10, replace = T)
)

df <- data.frame(
  x = rnorm(10),
  text = sample(c("Ana", "Eduardo"), size = 10, replace = T)
)
```

```
str(tib)
```

```
tibble [10 x 2] (S3: tbl_df/tbl/data.frame)
 $ x    : num [1:10] 0.172 0.315 0.119 -0.155 -0.165 ...
 $ text: chr [1:10] "Eduardo" "Ana" "Eduardo" "Eduardo" ...
```

```
str(df)
```

```
'data.frame': 10 obs. of 2 variables:
 $ x    : num 0.0639 -0.4522 0.7528 -1.3353 1.454 ...
 $ text: Factor w/ 2 levels "Ana","Eduardo": 2 2 2 1 2 2 1 1 2 1
```

Uma última característica de um tibble, é que ele lhe permite criar colunas com nomes que não respeitam as regras usuais do R. Por exemplo, não é permitido criar variáveis que possuam um nome que se inicia por um número, ou então, que possuam algum tipo de espaço ao longo dele. Mas dentro de um tibble, você não possui tais restrições. No exemplo abaixo, eu tento ultrapassar essa regra na função `data.frame()`, e ela acaba preenchendo o espaço no nome, com um ponto (`.`), e também coloca uma letra qualquer antes do número da coluna “10_janeiro”, enquanto em um tibble, isso não ocorre. Entretanto, mesmo que você possua essa liberdade em um tibble, ao se referir a essas colunas que não se encaixam nas regras do R, você terá de contornar o nome dessas colunas, com acentos graves (```).

```
data_frame <- data.frame(
  "Nome coluna" = rnorm(10),
  "10_janeiro" = rnorm(10)
)
```

```
tibble <- tibble(
  "Nome coluna" = rnorm(10),
  "10_janeiro" = rnorm(10)
)
```

```
head(data_frame, 10)
```

```
##      Nome.coluna X10_janeiro
## 1      2.0205593 -0.69919119
## 2     -0.3188909 -0.50518742
## 3     -1.5154911  0.62001069
## 4      0.4867145  0.54138829
## 5     -2.6753178 -0.84358430
## 6     -0.8957192  0.95547327
## 7      1.3036159  0.56695523
## 8      2.7856373 -0.54689044
## 9     -0.9902427  0.39354953
## 10     2.0933354 -0.06963645
```

```
tibble
```

```
## # A tibble: 10 x 2
##   `Nome coluna` `10_janeiro`
##   <dbl>         <dbl>
## 1      0.360      -1.29
## 2      0.291     -0.614
## 3     -1.01       0.227
## 4     -0.438     0.680
## 5     -0.119     0.498
## 6     -0.999     0.780
## 7     -0.0834    0.418
## 8      0.301     0.641
## 9      0.573    -0.481
## 10     1.14     0.0412

tibble$`10_janeiro`

## [1] -1.28820605 -0.61434222  0.22691214  0.67964702  0.49831633
## [6]  0.78000346  0.41829525  0.64144433 -0.48074841  0.04119007
```

Portanto, os tibble's foram criados com o intuito de manter as funcionalidades importantes de um `data.frame`, e ao mesmo tempo, eliminar comportamentos que hoje são desnecessários ou ineficientes. Em resumo, um tibble é uma estrutura preguiçosa. Pois ele nunca converte implicitamente os seus dados para algum outro tipo, ele não altera o nome de suas colunas, e ele também não sobrecarrega o seu console com linhas e linhas de resultados, lhe mostrando apenas o necessário.

2.5 Tipos de dados

Como foi destacado anteriormente, além das estruturas de dados, o R possui os tipos de dados. Tipos esses que dizem respeito a forma como o R está interpretando os seus dados, em um dado momento. Os cinco tipos de dados básicos da linguagem são:

1. `character`: valores de texto ou caracteres.
2. `double`: valores numéricos incluídos no conjunto dos números reais.
3. `integer`: valores numéricos incluídos no conjunto de números inteiros, ou basicamente, números sem casas decimais.
4. `logical`: valores `TRUE` (verdadeiro) e `FALSE` (falso), resultantes de testes lógicos.
5. `complex`: valores em números complexos.

Há vários outros tipos de dados mais complexos, como datas (`Date`) e fatores (`factor`), que são construídos a partir desses tipos básicos da linguagem. Tendo isso em mente, o único tipo básico que não irei abordar nesta seção, será o tipo `complex`, pois é um tipo muito específico e extremamente raro na linguagem.

2.5.1 Textos e caracteres

Você geralmente utiliza valores em texto em quase todos os instantes de sua análise, seja para criar rótulos de seus valores numéricos, indicando a qual indicador, região ou grupo aqueles valores se referem, ou então para criar rótulos, títulos e subtítulos elegantes para o seu gráfico. Todo valor em texto no R, deve ser fornecido entre aspas (simples - ', ou duplas - "), sendo essa uma convenção utilizada em quase todas as linguagens de programação, e no R não é diferente. No caso do R, esta convenção se torna ainda mais importante, pois ela também serve para diferenciar valores em texto dos nomes de objetos.

Quando queremos acessar os valores que estão dentro de um objeto, nós escrevemos o nome deste objeto no console. Mas quando estamos fornecendo um valor de texto ao R, é muito comum que nos esqueçamos de contorná-lo com aspas. Como resultado, o R acaba procurando por um objeto que possua um nome igual a este valor, e caso o R não encontre um objeto com esta característica, ele acaba lhe retornando um erro indicando que ele não encontrou um objeto com este nome em sua sessão. Além disso, se este valor que você está dando ao R possuir algum espaço, o R irá lhe retornar um erro um pouco diferente, dizendo que o símbolo o qual você inseriu no console, é inválido. Por isso, você deve lembrar de contornar esse valor por aspas, caso você deseje que ele seja interpretado como um texto simples.

```
> 0_ano_tem_365_dias
## Erro: objeto '0_ano_tem_365_dias' não encontrado
```

```
> 0 ano tem 365 dias
## Erro: unexpected symbol in "0 ano"
```

```
> "0 ano tem 365 dias"
## [1] "0 ano tem 365 dias"
```

Isso não quer dizer que você precisa escrever na mão todos os valores contornados por aspas. Na maioria das vezes, quando você importar as suas bases de dados, o R irá automaticamente converter os seus textos para character's. Caso ele converta de forma incorreta esses valores para algum outro tipo de dado, você pode facilmente corrigir isso, obrigando-o a converter esses valores para textos com a função `as.character()`.

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

```
vetor_d <- c(2.25, 4.1, 7.8)
```

```
as.character(vetor_l)
```

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

```
as.character(vetor_d)
```

```
## [1] "2.25" "4.1" "7.8"
```

Todos os outros tipos que citamos anteriormente podem ser convertidos para textos, pois este é o tipo mais flexível de todos. O motivo disto é simples: nós não podemos escrever textos (ou palavras), em números, digo, qual seria o número correspondente à letra “a” ? 1 ? 2 ? 3 ? ...; mas nós podemos escrever números, datas, nomes, fatores, TRUE e FALSE em textos. Basta contornar todos esses diferentes tipos e valores por aspas, que o R irá interpretá-los como textos (character), ao invés de seus tipos originais.

2.5.2 Números reais

Quase sempre que estiver trabalhando com dados numéricos, esses dados estarão sendo interpretados como double's, pois este tipo básico abarca todo o conjunto dos números reais. E como o conjunto de números inteiros (integer) está incluso no conjunto dos números reais, quando você insere um número inteiro, ou um número sem casas decimais no console, ele será interpretado inicialmente pelo R como um número real (double).

Dito de outra forma, se eu for ao console, e inserir apenas o número 10, o R estará interpretando este 10 como um double, e não como integer, mesmo que ele esteja lhe mostrando no console este número sem casas decimais. É como se este 10, fosse na verdade para o R algo como 10,000000000000... No exemplo abaixo, eu utilizo a função `is.integer()` para perguntar ao R, se ele está interpretando este valor como um integer, e como esperávamos a função nos retorna um FALSE, indicando que não se trata de um número inteiro.

```
# O R está basicamente interpretando  
# este 10 como 10.0000000000, mesmo  
# que ele te mostre  
10
```

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

```
is.double(10)
```

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

```
is.integer(10)
```

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

Vale destacar, que o R é uma linguagem centralizada nos padrões americanos, e que portanto, utiliza o ponto para definir casas decimais, ao invés da vírgula que nós brasileiros utilizamos. Logo, se você quer criar um vetor de números decimais, por exemplo, você deve definir as casas decimais de seus valores, através de pontos, e as vírgulas vão servir apenas para separar esses valores no vetor.

```
c(1.24, 2.25, 3.62381, 7.05)
```

```
## [1] 1.24000 2.25000 3.62381 7.05000
```

Você pode converter um vetor, ou um conjunto de valores para o tipo `double`, através da função `as.double()`. Basta fornecer o vetor, ou o conjunto de valores que deseja converter, à função:

```
vetor_1 <- c(TRUE, FALSE, TRUE, FALSE)
```

```
as.double(vetor_1)
```

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

2.5.3 Números inteiros

O tipo `integer` abarca o conjunto dos números inteiros, ou basicamente todos os números sem casas decimais. Você utilizará muito este tipo, quando estiver utilizando sequências numéricas, seja para extrair partes de um objeto com a função `[]`, ou gerando um índice para as linhas de sua tabela. Como vimos na seção anterior, caso você insira um número sem casas decimais no console, o R irá interpretar inicialmente este número como um `double`.

Assim sendo, você tem três formas de criar um `integer` no R. A primeira é inserindo um `L` maiúsculo após o número que está criando. A segunda, é transformando o seu vetor de números (que se encontra no tipo `double`) para `integer`, através da função `as.integer()`. A terceira, seria através de funções que lhe retornam por padrão este tipo de dado, sendo o principal exemplo, a função `:` que lhe retorna por padrão uma sequência de `integer`'s. Podemos confirmar se os números criados são de fato `integer`'s, usando a função `is.integer()`.

```
c(1L, 2L, 3L, 10L)
```

```
## [1] 1 2 3 10
```

```
as.integer(c(1, 2, 10, 1.5))
```

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

```
is.integer(1:10)
```

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

2.5.4 Valores lógicos

Este talvez seja o tipo básico que você esteja mais curioso sobre. Você já deve ter percebido que temos apenas dois valores possíveis dentro deste tipo, que são verdadeiro - TRUE, e falso - FALSE. Você irá utilizar muito este tipo para filtrar linhas de seu `data.frame`, para preencher uma coluna de rótulos, ou para identificar valores “não disponíveis” e *outliers* de sua base.

Você possui duas formas de obter esses valores no R. A primeira, é escrevê-los na mão, podendo também se referir apenas a primeira letra maiúscula de cada um, ao invés de escrever toda a palavra. A segunda e principal forma, é através de testes lógicos. No exemplo abaixo, eu estou criando um vetor com 5 elementos, e em seguida, peço ao R que me diga se cada elemento deste vetor é maior do que 5. Vemos que apenas o terceiro e o quarto elemento deste vetor, são maiores do que 5.

```
vetor <- c(0.5, 2.45, 5.6, 7.2, 1.3)
```

```
vetor > 5
```

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

O que acabamos de fazer acima, se trata de um teste lógico, pois estamos testando uma hipótese (maior do que 5) sobre cada um dos elementos deste vetor. Como resultado, o R lhe retorna um vetor com o mesmo comprimento do primeiro, porém agora, este vetor está preenchido com TRUE's e FALSE's, lhe indicando quais dos elementos do primeiro vetor se encaixam na hipótese que você definiu.

Este vetor contendo apenas valores lógicos, não é tão útil em sua singularidade. Porém, ao utilizarmos ele sobre a função `[]`, podemos utilizar o sistema que mencionei anteriormente, chamado de *logical subsetting*, que é uma forma extremamente útil de extrairmos partes de um objeto. A ideia, é extrairmos qualquer elemento deste objeto que possua um valor TRUE correspondente em um teste lógico específico que podemos definir. Consequentemente, poderíamos utilizar o teste anterior que criamos, para extrair todos os elementos do vetor, que são maiores do que 5, desta forma:

```
vetor[vetor > 5]
```

```
## [1] 5.6 7.2
```

2.6 Coerção no R

Quando discuti sobre vetores e sua principal propriedade (**vetores podem manter apenas um tipo de dado dentro dele**), eu mencionei que caso você tentasse burlar essa regra, o R automaticamente converteria todos os valores para um único tipo de dado. Este processo é usualmente chamado por *coercion*, ou coerção, e iremos explicar como ele funciona nesta seção.

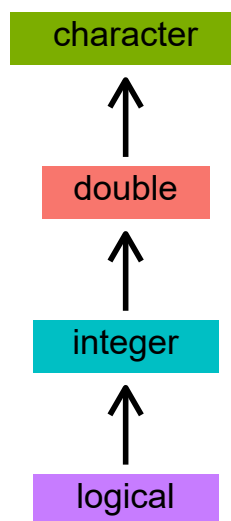
Você geralmente não provoca este evento propositalmente, mas ele pode ocorrer ao importar bases onde seus dados não seguem um padrão uniforme, ou quando seus valores vazios são representados

por algum caractere especial. Agora, se os números e valores da sua base estão bem formatados, e os valores vazios são realmente vazios, você dificilmente enfrentará este problema. Mas é útil conhecê-lo, pois este evento gera confusão em muitos usuários, em especial ao perceberem que seus números estavam o tempo todo sendo interpretados como texto (`character`).

Este processo de coerção ocorre apenas sobre vetores atômicos. Porém, lembre-se que todas as outras estruturas são construídas a partir desses vetores, ou todas as outras estruturas podem conter esses vetores dentro delas. Logo, uma coluna de seu `data.frame`, ou toda uma matriz, podem ser convertidos para o tipo de dado errado, independentemente de você ter ou não requisitado por tal transformação.

Quando o processo de coerção ocorre, o R irá transformar os dados para o tipo mais flexível, seguindo uma espécie de árvore, que está referenciada na figura 2.10. Você pode ver que o tipo `character`, está no topo da árvore, e portanto, é o tipo mais flexível de todos, enquanto o `logical` que está na base, é o tipo mais restrito de todos. Isso significa, que se você criar um vetor com valores `integer` e `logical`, todos esses valores serão convertidos para `integer`'s. Se for um vetor com valores `integer` e `character`, esses valores serão convertidos para `character`'s. E assim por diante. Ou seja, você sabe para qual tipo esse vetor será convertido, ao olhar para os dois tipos que estão sendo misturados neste vetor, e identificar o tipo mais flexível dos dois.

Figura 2.11: Árvore de tipos para coerção



Fonte: Elaboração própria do autor.

Isto não significa que você não pode criar um vetor de `logical` a partir de qualquer um dos outros tipos básicos. Mas para realizar essa transformação, você terá que pedir explicitamente por ela, através da função `as.logical()`. Se o seu vetor se encontra no tipo `double` ou `integer`, valores que são iguais a 0, serão convertidos para `FALSE`, e qualquer outro valor diferente de 0 será convertido

para TRUE. Mas se o seu vetor se encontra no tipo character, apenas textos explícitos dos valores lógicos (FALSE e TRUE), podem ser convertidos.

Após toda essa leitura, você deve ter compreendido, que temos funções `is.*` e funções `as.*` para cada um dos quatro tipos básicos. As funções `is.*` servem para confirmar se os dados alocados em um objeto, estão ou não em um determinado tipo de dado. Ou seja, se eu quero saber se uma coluna de meu `data.frame` está no tipo `double`, eu utilizo a função `is.double()` sobre esta coluna. Já as funções `as.*` servem para converter explicitamente os valores para um tipo de dado específico. Portanto, se eu tenho um vetor com `double's` e quero transformá-los em `character's`, eu forneço este vetor à função `as.character()`. Quando uma função `as.*` encontrar um elemento deste vetor, que ela não consegue converter para o tipo especificado, a função acaba inserindo um NA (valor não disponível) no lugar deste elemento.

```
vetor <- c(0, 1, 0.5, -2, 20)

as.character(vetor)

## [1] "0"    "1"    "0.5"  "-2"   "20"

as.logical(vetor)

## [1] FALSE  TRUE   TRUE   TRUE   TRUE

as.integer(vetor)

## [1] 0  1  0 -2 20

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

as.logical(vetor)

## [1] NA NA NA
```

2.7 Subsetting

As operações de *subsetting*, são extremamente importantes no R, e você irá utilizá-las com grande frequência ao longo de seu trabalho. Ao longo das seções de [Estruturas de Dados](#), eu dei exemplos sobre como utilizar o *subsetting* com cada tipo de estrutura. Tendo isso em mente, essa seção busca explicitar (ou formalizar) algumas características importantes dessas operações. Como o próprio nome dá a entender, as operações de *subsetting* servem para extrairmos ou modificarmos *subsets* (partes) de seus objetos ([R CORE TEAM, 2020b](#)). Como vimos anteriormente, essas operações são realizadas pelas funções `[]` e `[[`.

Para utilizar a função `[]`, você precisa abrir um par de colchetes (`[]`) após o nome do objeto (ou função) com o qual está trabalhando. Já para a função `[][]`, você necessita abrir dois pares de colchetes (`[][]`) após o nome (ou função) com o qual você está trabalhando. Também já vimos ao longo das seções de **Estruturas de Dados**, que para extrairmos partes de estruturas unidimensionais como vetores e listas, precisamos de apenas um índice, ou de um único conjunto de índices. Mas para extrairmos partes de estruturas bidimensionais, como matrizes e `data.frame`'s, precisamos de dois índices, ou de dois conjuntos de índices.

Além disso, lembre-se que como definimos anteriormente, as listas são estruturas especiais, pois podem conter diversas outras estruturas em seus elementos. Portanto, apesar das listas serem estruturas unidimensionais, elas podem conter outras estruturas bidimensionais dentro delas. Por isso, caso você esteja interessado em extrair partes de uma estrutura bidimensional, que está dentro de algum elemento de uma lista, por exemplo, você irá precisar de uma combinação entre um único índice (para acessar o elemento da lista) e outros dois conjuntos de índices (para acessar uma parte específica da estrutura bidimensional).

2.7.1 Principais diferenças entre as funções `[]` e `[][]`:

- 1) A função `[]` pode trabalhar com todas as dimensões disponíveis de um objeto. As dimensões disponíveis dependem da estrutura em que esse objeto se encontra. Enquanto isso, a função `[][]` pode trabalhar com apenas uma dessas dimensões disponíveis.
- 2) A função `[]` permite você extrair um conjunto de elementos (ou seções) de um objeto (Ex: da 1° a 100° linha de um `data.frame`; os elementos 4, 5 e 8 de um vetor; do 3° ao 6° elemento de uma lista). Já a função `[][]` lhe permite extrair uma única parte, ou um único elemento de um objeto (Ex: o 5° elemento de uma lista; a 2° coluna de um `data.frame`; o 10° elemento de um vetor).
- 3) A função `[]` geralmente lhe retorna um resultado na mesma estrutura de seu objeto original. Em outras palavras, se você utilizar a função `[]` sobre uma lista, ela irá lhe retornar uma lista como resultado. Já a função `[][]`, geralmente lhe retorna um resultado em uma estrutura diferente. Dito de outra forma, se você utilizar a função `[][]` sobre um `data.frame`, por exemplo, ela geralmente vai lhe retornar um vetor como resultado.

2.7.2 Dimensões disponíveis em *subsetting*

A estrutura em que um objeto se encontra, define as dimensões que estão disponíveis para as funções `[]` e `[][]`. Logo, se você está trabalhando com um `data.frame`, por exemplo, você possui duas dimensões (linhas e colunas) com as quais você pode trabalhar com a função `[]`. Mas se você está trabalhando com uma estrutura unidimensional, como um vetor atômico, você terá apenas uma única dimensão (os elementos desse vetor) para trabalhar em ambas às funções de *subsetting* (`[]` e `[][]`).

Uma das diferenças básicas entre as funções `[]` e `[][]`, se encontra no número de dimensões com as quais elas podem trabalhar. A função `[]`, seria uma forma mais “geral” de *subsetting*, pois ela pode

trabalhar com todas as dimensões disponíveis segundo a estrutura que um objeto se encontra. Já a função `[[`, representa uma forma mais restritiva de *subsetting*, pois ela trabalha em geral com apenas uma única dimensão de seu objeto (independentemente de qual seja a sua estrutura).

Portanto, se temos uma estrutura bidimensional como um `data.frame`, a função `[` pode trabalhar com as suas duas dimensões (linhas e colunas). Porém, a função `[[` pode trabalhar apenas com uma dessas dimensões, sendo no caso de `data.frame`'s, a dimensão das colunas. Agora, quando estamos trabalhando com uma estrutura unidimensional, como nós possuímos apenas uma dimensão (elementos) disponível, não há diferença entre as funções `[` e `[[` no sentido estabelecido anteriormente. De qualquer maneira, a função `[` continuará sendo a forma mais geral e flexível de *subsetting* para objetos unidimensionais. Pois a função `[` lhe permite selecionar um conjunto, ou uma sequência de elementos de uma estrutura unidimensional, enquanto que com a função `[[`, você poderá selecionar apenas um único elemento dessa estrutura. Um resumo das dimensões disponíveis em cada estrutura, se encontra na tabela 2.1.

Tabela 2.1: Resumo das dimensões disponíveis em cada estrutura

Estrutura	Tipo	[]	[[]]
Vetor	Unidimensional	elemento	elemento
Lista	Unidimensional	elemento	elemento
Matriz	Bidimensional	linha e coluna	elemento
data.frame	Bidimensional	linha e coluna	coluna

Fonte: Elaboração própria do autor.

Tabela 2.2: Notação matemática das dimensões disponíveis em cada estrutura

Estrutura	Notação	[]	[[]]
Vetor	V_e	[e]	[[e]]
Lista	L_e	[e]	[[e]]
Matriz	$M_{i,j}$	[i,j]	[[e]]
data.frame	$DF_{i,j}$	[i,j]	[[j]]

Fonte: Elaboração própria do autor.

Nós também podemos ver essas diferenças entre as dimensões disponíveis em cada estrutura e para cada função de *subsetting*, sob uma perspectiva mais matemática, ao formar uma notação matemática de cada estrutura, incluindo subscritos que representem as suas respectivas dimensões. Essa visão está exposta na tabela 2.2. Por exemplo, pegando um `data.frame` chamado *DF*, com

i linhas e j colunas ($DF_{i,j}$), temos que o comando `DF[2,4]` busca extrair o valor (ou valores) localizados na 2ª linha da 4ª coluna da tabela. Por outro lado, considerando-se uma lista chamada L , contendo e elementos (L_e), o comando `L[[4]]`, traz como resultado, o 4º elemento dessa lista.

2.7.3 Tipos de índices

Os índices que você fornece às funções `[]` e `[[ ]]`, podem ser de três tipos: 1) índices de texto - character; 2) índices numéricos - integer; 3) índices lógicos - logical. Logo abaixo, temos um exemplo do uso de índices numéricos sobre um vetor qualquer. Lembre-se que no caso de vetores, nós podemos utilizar um único índice para extrairmos um único valor do objeto em questão, e nós utilizamos dois ou mais índices, para extrairmos um conjunto de valores deste mesmo vetor.

```
vec <- c(2.2, 1.3, 4.5, 3.7, 5.2)
```

```
vec[4]
```

```
## [1] 3.7
```

```
vec[1:4]
```

```
## [1] 2.2 1.3 4.5 3.7
```

```
vec[c(3,5,1)]
```

```
## [1] 4.5 5.2 2.2
```

Para utilizar um índice de texto (character), o objeto sobre o qual você está trabalhando, deve ser uma estrutura nomeada. Todas as estruturas (vetor, lista, matriz e `data.frame`) permitem o uso de nomes, que você pode acessar e definir através de funções como `colnames()`, `row.names()` e `names()`. Sendo que algumas estruturas, mais especificamente os `data.frame`'s, vão sempre nomear automaticamente os seus elementos. Ou seja, você sempre poderá utilizar um índice de texto em um `data.frame`, para selecionar alguma de suas colunas. Pois mesmo que você se esqueça de nomear alguma coluna, ao criar o seu `data.frame`, a função que cria essa estrutura irá automaticamente criar um nome qualquer para cada coluna não nomeada.

```
df <- data.frame(
  id = LETTERS[1:10],
  nome = "Ana",
  valor = rnorm(10),
  "Belo Horizonte"
)
```

```
df
```

```
##      id nome      valor X.Belo.Horizonte.
## 1    A  Ana  0.43667038    Belo Horizonte
## 2    B  Ana -1.47896442    Belo Horizonte
## 3    C  Ana  1.87511826    Belo Horizonte
## 4    D  Ana -1.46671150    Belo Horizonte
## 5    E  Ana  0.05478447    Belo Horizonte
## 6    F  Ana  1.24923800    Belo Horizonte
## 7    G  Ana -0.53067035    Belo Horizonte
## 8    H  Ana  0.44917245    Belo Horizonte
## 9    I  Ana -0.59571461    Belo Horizonte
## 10   J  Ana -0.82987550    Belo Horizonte
```

```
colnames(df)[4] <- "cidade"
```

```
df[["cidade"]]
```

```
## [1] "Belo Horizonte" "Belo Horizonte" "Belo Horizonte" "Belo Horizonte"
## [5] "Belo Horizonte" "Belo Horizonte" "Belo Horizonte" "Belo Horizonte"
## [9] "Belo Horizonte" "Belo Horizonte"
```

```
df[c("id", "valor")]
```

```
##      id      valor
## 1    A  0.43667038
## 2    B -1.47896442
## 3    C  1.87511826
## 4    D -1.46671150
## 5    E  0.05478447
## 6    F  1.24923800
## 7    G -0.53067035
## 8    H  0.44917245
## 9    I -0.59571461
## 10   J -0.82987550
```

```
df[["valor"]]
```

```
## [1] 0.43667038 -1.47896442 1.87511826 -1.46671150 0.05478447
## [6] 1.24923800 -0.53067035 0.44917245 -0.59571461 -0.82987550
```

```
df[["nome"]]
```

```
## [1] "Ana" "Ana" "Ana" "Ana" "Ana" "Ana" "Ana" "Ana" "Ana" "Ana"
```

Em outras estruturas como um vetor, nomes não são atribuídos automaticamente a cada um de seus elementos, e por isso, você deve nomear os elementos deste vetor, para que você seja capaz de

utilizar um índice de texto nele. Para isso, basta igualar esses elementos a um valor em texto (valor entre aspas) que representa esse nome, como no exemplo abaixo:

```
vec <- c("a" = 1, "b" = 2, "c" = 3, "d" = 4)

vec["c"]

## c
## 3

vec[c("a", "c", "b")]

## a c b
## 1 3 2

vec[["b"]]

## [1] 2
```

Por último, os índices lógicos (TRUE ou FALSE) são extremamente úteis em diversas aplicações, especialmente quando desejamos realizar um *subsetting* mais “complexo”. Porém, pelo fato de que a função `[[` nos permite extrair apenas uma única parte de um objeto, os índices lógicos são de certa forma inúteis com essa função. Portanto, sempre que utilizar índices do tipo lógico para selecionar os seus dados, você muito provavelmente quer utilizá-los com a função `[`. Por padrão, as funções `[` e `[[`, vão extrair todas as partes de um objeto, que possuírem um valor TRUE correspondente.

Portanto, no exemplo abaixo, caso eu utilize o vetor lógico `vlog`, para selecionar valores do vetor `vec`, a função `[` irá selecionar o 2º, 3º e 5º valor do vetor `vec`. Pois são essas as posições no vetor `vlog` que contém TRUE's. Porém, a principal forma de gerarmos esses vetores lógicos a serem utilizados na função `[`, é através de testes lógicos. Por exemplo, podemos testar quais valores do vetor `vec`, são maiores do que 3, através do operador lógico `>` (maior que).

```
vec <- c(2.2, 1.5, 3.4, 6.7, 8.9)

vlog <- c(FALSE, TRUE, TRUE, FALSE, TRUE)

vec[vlog]

## [1] 1.5 3.4 8.9

vec[vec > 3]

## [1] 3.4 6.7 8.9
```

O R possui vários operadores lógicos diferentes, e o operador > é apenas um deles. Um outro operador muito conhecido, é o de negação "!". Este operador é utilizado, quando você deseja inverter um teste lógico, ou de certa forma, inverter o comportamento da função [quando fornecemos índices lógicos. O que o operador ! faz na verdade, é inverter os valores de um vetor lógico. Logo, se eu aplicar este operador ao vetor vlog, esse será o resultado:

```
!vlog
```

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

Portanto, os valores que antes eram TRUE, passam a ser FALSE, e vice-versa. Por isso, ao utilizarmos o operador ! sobre um teste lógico qualquer, nós invertemos o teste em questão. Pois o operador ! inverte os valores do vetor lógico resultante desse teste. Com isso, se eu utilizar esse operador sobre o teste anterior, onde testamos quais valores do vetor vec são maiores do que 3, nós estaremos efetivamente testando a hipótese contrária, de que esses valores são menores ou iguais a 3. Vale ressaltar, que esse operador deve ser posicionado antes do objeto que você deseja inverter, ou antes do teste lógico a ser realizado.

```
vec[!vec > 3]
```

```
## [1] 2.2 1.5
```

Um uso muito comum deste operador, é em conjunto com a função is.na(). Essa função, aplica um teste lógico sobre cada valor de um vetor, testando a hipótese de que esse valor se trata de um valor não-disponível (NA). Por isso, caso o valor em questão, seja de fato um valor não-disponível, a função is.na() irá retornar um TRUE correspondente, caso contrário, a função vai lhe retornar um FALSE. Logo, caso eu utilize a função is.na() dentro da função [, estaremos selecionando todos os valores não-disponíveis de um vetor. Porém, é muito mais comum que as pessoas queiram fazer justamente o contrário, que é eliminar esses valores não-disponíveis de seus dados. Por essa razão, é muito comum que se utilize o operador ! em conjunto com a função is.na(), pois dessa forma, estaremos selecionando justamente os valores que se encaixam na hipótese contrária a testada por is.na().

```
vec <- c(2.2, 1.3, NA_real_, NA_real_, 2.5)
```

```
vec
```

```
## [1] 2.2 1.3 NA NA 2.5
```

```
vec[is.na(vec)]
```

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



```
vec[!is.na(vec)]
```

```
## [1] 2.2 1.3 2.5
```

Vamos pensar no caso de um `data.frame`. Como definimos anteriormente, temos duas dimensões com as quais podemos trabalhar na função `[]`, com este tipo de estrutura. Podemos por exemplo, utilizar o operador `!` e a função `is.na()` sobre a dimensão das linhas desse `data.frame`. Dessa forma, podemos eliminar todas as linhas dessa tabela, que possuam algum valor não-disponível em uma coluna. Veja o exemplo abaixo, em que uma tabela chamada `df`, contém três valores não-disponíveis na coluna `valor`.

```
df <- data.frame(
  id = LETTERS[1:8],
  valor = c(1.2, 2.5, NA_real_, 5.5, NA_real_, NA_real_, 3.5, 1.3),
  nome = sample(c("Ana", "Luiza", "João"), size = 8, replace = TRUE)
)
```

```
df
```

```
##   id valor  nome
## 1  A   1.2 João
## 2  B   2.5  Ana
## 3  C    NA Luiza
## 4  D   5.5 Luiza
## 5  E    NA João
## 6  F    NA  Ana
## 7  G   3.5 Luiza
## 8  H   1.3 João
```

```
nao_e_NA <- !(is.na(df$valor))
```

```
df[nao_e_NA, ]
```

```
##   id valor  nome
## 1  A   1.2 João
## 2  B   2.5  Ana
## 4  D   5.5 Luiza
## 7  G   3.5 Luiza
## 8  H   1.3 João
```

2.7.4 O operador `$` e a estrutura do resultado

Você provavelmente se lembra do operador `$`, que se trata de um atalho à função `[]`. Porém, você talvez tenha percebido também, que utilizamos o operador `$` apenas em estruturas nomeadas. Logo,

apesar de o operador `$` ser um “irmão” da função `[[`, ele não herda todas as características dessa função. Por exemplo, nós não podemos utilizar índices numéricos ou lógicos com este operador, para selecionarmos alguma parte de um objeto. Isto significa, que o operador `$` se trata de uma versão ainda mais restrita de *subsetting*, em relação à função `[[`. As únicas estruturas nomeadas com as quais este operador funciona, são listas e `data.frame`'s. Em outras palavras, mesmo que você nomeie os elementos de um vetor atômico, você não poderá utilizar o operador `$` para selecionar um desses elementos.

```
vec <- c("a" = 2.5, "b" = 4.3, "c" = 1.2)
```

```
vec$a
```

```
Error in vec$a : $ operator is invalid for atomic vectors
```

Dentre as características da função `[[` herdadas pelo operador `$`, está o fato de que este operador pode trabalhar apenas com uma dimensão de um objeto. Em listas, podemos utilizar o operador `$` para selecionarmos algum dos elementos nomeados dessa lista. Já em `data.frame`'s, o operador `$` pode ser utilizado para selecionarmos uma das colunas desse `data.frame`³.

Um outro ponto a ser discutido, é que tanto o operador `$`, quanto a função `[[`, geram um resultado em uma estrutura diferente da estrutura do objeto original. Ou seja, quando realizamos um *subsetting* por meio desses operadores, o resultado geralmente possui uma estrutura com menos componentes do que a estrutura do objeto original, de onde estamos retirando esta parte. Dito de outra forma, se utilizarmos o operador `$`, ou a função `[[` para selecionarmos a coluna valor do `data.frame` `df` abaixo, o resultado de ambas as funções, serão um vetor atômico contendo os valores dessa coluna, e não um `data.frame` contendo apenas a coluna valor.

Logo, o uso da função `[[` (ou do operador `$`) sobre `data.frame`'s, vão lhe trazer a coluna (ou o elemento) em si do `data.frame`, e não um novo `data.frame` contendo essa coluna. Podemos confirmar isso, com o uso da função `str()`, que nos traz um resumo da estrutura de um objeto. Perceba nos exemplos abaixo, que em ambos os casos, o resultado da função `str()` está nos dizendo que o objeto resultante do uso de `$` ou de `[[`, se trata de um vetor atômico contendo dados do tipo numérico (`num`).

```
df <- data.frame(  
  id = LETTERS[1:10],  
  valor = rnorm(10),  
  nome = sample(c("Ana", "Luiza", "João"), size = 10, replace = TRUE)  
)
```

```
str(df$valor)
```

³Lembre-se que no fundo, `data.frame`'s são listas, com a propriedade de que todos os elementos dessa lista, devem possuir o mesmo número de linhas. Portanto, se cada coluna desse `data.frame` representa um elemento da lista que forma esse `data.frame`, ao utilizarmos o operador `$`, também estaríamos selecionando um “elemento”, que se traduz em uma coluna do `data.frame`.

```
## num [1:10] -0.561 1.668 0.457 -1.25 -0.225 ...
```

```
str(df[["valor"]])
```

```
## num [1:10] -0.561 1.668 0.457 -1.25 -0.225 ...
```

Essa característica é definida em detalhes no capítulo 4 de [Wickham \(2015a\)](#). Sendo exatamente esta característica, que eu estava querendo destacar na figura 2.7, quando estávamos descrevendo as listas. Se você utilizar a função `[` para selecionar um elemento de uma lista, o resultado será uma nova lista contendo esse elemento. Mas se você utilizar a função `[[` para fazer este trabalho, o resultado será apenas o elemento em si.

Você pode entender essa característica como uma “simplificação do resultado”, como se as funções `[[` e `$` gerassem um resultado em uma estrutura mais simples do que a do objeto original. Porém, eu creio que essa é uma forma equivocada de se enxergar esse sistema, pois estruturas não são usualmente comparadas em níveis de complexidade, mas sim por suas propriedades e características. Por isso, uma forma mais útil e fiel de se enxergar essa característica, é através da representação apresentada pela figura 2.7, onde através da função `[[`, podemos selecionar o elemento em si de uma lista, e não uma lista contendo este elemento. Além disso, uma outra forma útil de enxergarmos essa característica no resultado das funções `[[` e `$`, é como uma forma de eliminarmos componentes da estrutura do objeto original. Em outras palavras, podemos enxergar o operador `$` ou a função `[[`, como uma forma de gerarmos um resultado com menos componentes do que a estrutura do objeto original.

Por exemplo, se temos um `data.frame` chamado `df`, onde temos duas colunas simples (que são vetores atômicos), e em seguida, adicionamos duas novas colunas, uma contendo uma lista, e outra contendo um outro `data.frame` de duas colunas (`y` e `z`), nós temos uma estrutura razoavelmente complexa. Se utilizarmos a função `str()`, para nos fornecer um resumo da estrutura de `df`, vemos que esse objeto tem pelo menos três componentes: 1) os vetores representados pelas colunas `x` e `nome`; 2) os cinco elementos da lista alocada na coluna `lista`; 3) e as duas colunas contidas no `data.frame` da coluna `outro_df`.

```
df <- data.frame(
  x = rnorm(5),
  nome = "Ana"
)

df$lista <- list(1, 2, 3, 4, 5)
df$outro_df <- data.frame(y = rnorm(5), z = rnorm(5))

str(df)

## 'data.frame':    5 obs. of  4 variables:
## $ x          : num  -1.14 0.363 -1.235 0.573 0.362
```

```
## $ nome      : chr  "Ana" "Ana" "Ana" "Ana" ...
## $ lista     :List of 5
##   ..$ : num 1
##   ..$ : num 2
##   ..$ : num 3
##   ..$ : num 4
##   ..$ : num 5
## $ outro_df:'data.frame':  5 obs. of  2 variables:
##   ..$ y: num  -0.642 0.634 -0.496 -0.458 1.287
##   ..$ z: num  -0.653 -0.636 0.844 1.008 0.334
```

Caso eu utilize as funções `[[` e `$` para selecionarmos alguma das colunas de `df`, podemos aplicar novamente a função `str()` sobre o resultado, para compreendermos sua estrutura. Veja pelo exemplo abaixo, que o resultado da função `str()` nos descreve uma estrutura com menos componentes do que a estrutura original. Com isso, eu quero destacar que a estrutura desse resultado não necessariamente será menos “complexa” do que a original, mas sim que essa estrutura terá menos componentes. Portanto, pelo menos um dos componentes da estrutura original, será eliminado com o uso de `[[` ou de `$`.

```
str(df[["lista"]])
```

```
## List of 5
## $ : num 1
## $ : num 2
## $ : num 3
## $ : num 4
## $ : num 5
```

```
str(df[["outro_df"]])
```

```
## 'data.frame':  5 obs. of  2 variables:
## $ y: num  -0.642 0.634 -0.496 -0.458 1.287
## $ z: num  -0.653 -0.636 0.844 1.008 0.334
```

```
str(df$outro_df)
```

```
## 'data.frame':  5 obs. of  2 variables:
## $ y: num  -0.642 0.634 -0.496 -0.458 1.287
## $ z: num  -0.653 -0.636 0.844 1.008 0.334
```

2.8 Valores especiais do R

Na linguagem R, possuímos alguns valores especiais, que não apenas são tratados de maneira diferente em relação a outros valores, mas que também efetivamente alteram o comportamento de

algumas operações importantes na linguagem. Por exemplo, se você tentar dividir qualquer número por 0 no console, ao invés do R lhe retornar um erro, lhe indicando que essa divisão é indefinida, o console vai lhe retornar o valor Inf, que se refere a infinito (ou *infinite*). Por outro lado, de forma ainda mais estranha, se você tentar dividir 0 por ele mesmo, o console vai lhe retornar o valor NaN, que significa “*not a number*”, ou em outras palavras, que o valor resultante da divisão não é um número.

Esses são alguns exemplos de valores especiais que você pode adquirir. Porém, o valor especial mais comum, é o valor NA, que significa *not available*, ou “não-disponível”. Este valor geralmente é resultado de uma dessas duas situações: 1) ao importar a sua base de dados para o R, a linguagem vai preencher automaticamente todas as células em sua base que estiverem vazias, com um valor NA; 2) quando você executa (ou causa de maneira indireta) um processo de coerção, no qual o R não consegue realizar. Ou seja, se o R não souber como converter um valor específico, para o tipo de dado ao qual você requisitou, ele vai lhe retornar um valor NA correspondente a aquele valor.

Portanto, a primeira situação ocorre durante o processo de importação de dados, em todas as ocasiões em que você possuir alguma observação vazia na base de dados que você está importando. Logo, se em uma planilha do Excel, por exemplo, você possuir alguma célula vazia em sua tabela, ao importar essa planilha para o R, essas células vazias serão preenchidas com valores NA no R. Lembre-se que um valor NA indica uma observação não-disponível, o que significa que o valor correspondente aquela observação não pôde ser observado, ou não pôde ser registrado no momento de coleta dos dados.

Já a segunda situação, ocorre sempre quando o R não sabe como realizar o processo de coerção, pelo qual requisitamos, de uma forma lógica. Por exemplo, isso ocorre ao tentarmos converter valores de texto para números com `as.double()`. Pois o R não sabe como, ou não sabe qual a maneira mais adequada de se converter esses valores em texto para números. Por isso, a linguagem vai lhe retornar como resultado, valores NA.

Por que estamos falando desses valores especiais? Porque eles alteram o comportamento de certas operações importantes do R e, com isso, podem deixar você desorientado! Por exemplo, se você tentar calcular a soma de uma coluna (de um `data.frame`) que contém um valor NA, o resultado dessa operação será um valor NA. Da mesma forma, se a coluna possuir um valor NaN, o resultado dessa soma será um valor NaN. Para que isso ocorra, o valor especial pode estar em qualquer linha que seja, basta que ele ocorra uma única vez, que a sua soma não vai funcionar.

```
sum(c(1, 2, 3, NA, 4))
```

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

```
sum(c(1, 2, 3, NaN, 4))
```

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

Isso não significa que esses valores especiais serão uma dor de cabeça para você, pois cada um deles tem o seu propósito, e eles o cumprem muito bem. Mas é importante que você saiba do

quão especiais eles são, e dos efeitos que eles causam em certas operações no R. Com isso, se em alguma situação uma função lhe retornar um valor NA, quando ela deveria lhe retornar algum valor definido, ou se essa função se comportar de maneira inesperada, você pode desconfiar que algum valor especial presente em seus dados, possa ser a fonte de sua surpresa.

Em geral, todas as funções que são afetadas por esses valores especiais, como as funções `sum()` e `mean()`, possuem um argumento `na.rm`, que define se a função deve ignorar esses valores especiais em seus cálculos. Portanto, caso uma coluna de seu `data.frame` possua esses valores especiais, e você precisa ignorá-los durante o cálculo de uma soma, lembre-se de configurar este argumento para verdadeiro (`TRUE`).

```
sum(c(1, 2, 3, NA, 4), na.rm = TRUE)
```

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

Um outro tipo de operação importante que é afetada por esses valores especiais, são os testes lógicos. Como exemplo, vamos criar um teste lógico sobre os dados apresentados pela tabela `compras`. Nós temos nessa tabela, o nome da composição química dos principais remédios que estão em falta nos estoques de três grandes hospitais. Os três remédios presentes nessa tabela, são remédios bem comuns, como o valor AA que se refere à composição química da Aspirina (Ácido Acetilsalicílico).

```
compras <- structure(list(ano = c(2019, 2019, 2019, 2019, 2019, 2019, 2019,
2019, 2019, 2019), mes = c(2L, 4L, 5L, 6L, 8L, 8L, 10L, 10L,
10L, 12L), hospital1 = c("AA", NA, "dexametasona", "AA", NA,
"doxiciclina", NA, "AA", "doxiciclina", NA), hospital2 = c("AA",
"doxiciclina", "dexametasona", "dexametasona", "AA", NA, "dexametasona",
"AA", "dexametasona", "AA"), hospital3 = c("AA", "AA", "dexametasona",
NA, "dexametasona", "doxiciclina", "dexametasona", NA, NA, "AA"
)), row.names = 1:10, class = "data.frame")
```

```
compras
```

##	ano	mes	hospital1	hospital2	hospital3
## 1	2019	2	AA	AA	AA
## 2	2019	4	<NA>	doxiciclina	AA
## 3	2019	5	dexametasona	dexametasona	dexametasona
## 4	2019	6	AA	dexametasona	<NA>
## 5	2019	8	<NA>	AA	dexametasona
## 6	2019	8	doxiciclina	<NA>	doxiciclina
## 7	2019	10	<NA>	dexametasona	dexametasona
## 8	2019	10	AA	AA	<NA>
## 9	2019	10	doxiciclina	dexametasona	<NA>
## 10	2019	12	<NA>	AA	AA

Por exemplo, se nós quiséssemos identificar todas as linhas na tabela `compras`, em que a composição química da Aspirina (valor AA) aparece em pelo menos um dos hospitais (ou dito de outra forma, em pelo menos uma das colunas), poderíamos aplicar um teste lógico sobre a tabela `compras`. O teste lógico abaixo, serve para esse propósito, mas se olharmos para o resultado desse teste, podemos identificar que algo está errado.

```
teste <- compras$hospital1 == "AA" |
  compras$hospital2 == "AA" |
  compras$hospital3 == "AA"

teste

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

Perceba acima, que o teste lógico detectou com sucesso todas as linhas da tabela `compras`, que possuem um valor AA em pelo menos uma de suas colunas. Mais especificamente, as linhas de posição 1°, 2°, 4°, 5°, 8° e 10°. Porém, podemos também identificar, que para as linhas de posição 6°, 7° e 9° na tabela, o teste lógico `teste` nos retornou valores NA. Ou seja, ao invés do teste lógico nos retornar um valor FALSE, para as linhas que não possuem um valor AA ao longo de suas colunas, ele acaba nos retornando um valor NA, pelo simples fato de que temos um valor NA em pelo menos uma das colunas. Isso se torna um grande problema, a partir do momento em que desejamos filtrar a nossa tabela `compras`, ao fornecer o nosso vetor `teste`, à função de *subsetting*.

```
compras[teste, ]

##      ano mes hospital1 hospital2 hospital3
## 1  2019   2      AA      AA      AA
## 2  2019   4    <NA> doxiciclina      AA
## 4  2019   6      AA dexametasona    <NA>
## 5  2019   8    <NA>      AA dexametasona
## NA   NA  NA    <NA>    <NA>    <NA>
## NA.1 NA  NA    <NA>    <NA>    <NA>
## 8  2019  10      AA      AA    <NA>
## NA.2 NA  NA    <NA>    <NA>    <NA>
## 10 2019  12    <NA>      AA      AA
```

Portanto, o problema gerado pelos valores NA presentes no resultado do teste lógico, é que eles geram indiretamente um novo problema a ser resolvido. O objetivo principal está em identificar as linhas da tabela `compras`, que possuem um valor AA, em pelo menos uma de suas colunas, e filtrá-las da tabela. Porém, ao fornecermos esse vetor `teste` à função de *subsetting*, a função `[` acaba adicionando uma nova linha ao resultado, para cada valor NA presente no vetor `teste`. Logo, o resultado que era para ter 6 linhas, acaba tendo 9. Com isso, teríamos um novo trabalho de eliminar essas novas linhas de NA's, para chegarmos às linhas que queremos filtrar da nossa tabela `compras`.

Exercícios

Exercício 1

Em cada item desta questão, temos um simples `print()` de um objeto qualquer. Com base apenas nessa primeira imagem do objeto, tente identificar a estrutura (vetor, matriz, lista, `data.frame`) na qual esse objeto se encontra:

A) Objeto 1:

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
```

B) Objeto 2:

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

C) Objeto 3:

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

D) Objeto 4:

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

E) Objeto 5:

```
## $estado
## [1] "MG"
##
## $cidade
## [1] "Belo Horizonte"
##
```



```
## $n_municipios
## [1] 853
##
## $regiao
## [1] "Sudeste"
```

F) Objeto 6:

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

Exercício 2

Em cada item abaixo, você deve criar um teste lógico que seja capaz de testar as condições postas no enunciado. Em alguns itens, será fornecido o código necessário para que você crie certos objetos (como `v_seq`, `v_rep`, `lst`, etc.), pois os testes lógicos se baseiam nesses objetos, ou devem ser aplicados diretamente sobre esses objetos. Portanto, lembre-se de copiar o código fornecido pela questão, colar em seu console, e apertar Enter para recriar esses objetos em sua sessão do R.

- A)** Crie um teste lógico que seja capaz de identificar quais dos cinco objetos abaixo, são um vetor atômico.

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

- B)** Imagine que você receba em sua sessão do R, o objeto `lst` abaixo. Tente criar um teste lógico que possa confirmar se esse objeto é uma lista. Pelo código abaixo, você já sabe que este objeto é sim uma lista. Entretanto, nem sempre você terá acesso fácil ao código que criou certo objeto, ou, nem sempre você consegue prever que tipos de objetos serão criados a partir dos comandos escritos por outras pessoas. Por isso, quando você não conhece o *input* que você vai receber, é sempre importante se basear em teste lógicos que possam lhe assegurar que os seus objetos estão na estrutura desejada.

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

- C)** Utilizando a mesma lista `lst` do exercício acima, crie um teste lógico capaz de identificar se essa lista possui um item chamado “estado”. Primeiro, aplique o seu teste lógico sobre `lst`, e confira se o resultado do teste é `TRUE`. Em seguida, aplique esse mesmo teste lógico sobre a lista `lst_sem_estado` abaixo, e veja se o resultado do teste é de fato `FALSE`.

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

- D) Suponha que você possua a tabela `tab` abaixo. Crie um teste lógico que possa identificar se a coluna `total` é do tipo `double`.

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

- E) Utilizando a mesma tabela `tab` acima, crie um teste lógico que possa identificar se a tabela possui exatamente 10 linhas, E, se essa tabela possui uma coluna chamada “vendas”, E, se a 3ª coluna da tabela é do tipo `character`. Perceba no enunciado desta questão, os E’s separando cada condição a ser testada. Esses E’s estão indicando que essas condições são dependentes, ou, em outras palavras, elas precisam ser satisfeitas ao mesmo tempo.
- F) Se eu te der um número qualquer, referente a um ano específico (por exemplo, 2005 ou 1997), crie um teste lógico capaz de atestar se esse ano fornecido é um ano bissexto. Um ano bissexto é definido pelas seguintes condições: 1) a cada 4 anos, temos um ano bissexto; 2) a cada 100 anos, nós não devemos ter um ano bissexto; 3) a cada 400 anos temos um ano bissexto. Um detalhe: as últimas regras são mais importantes do que as primeiras, ou seja, a regra 3 prevalece sobre as regras 1 e 2, da mesma forma que a regra 2, prevalece sobre a regra 1. Caso essas definições não estejam muito claras, elas basicamente significam o seguinte: 1) o ano deve ser múltiplo de 4; 2) o ano não deve ser múltiplo de 100 a não ser que ele seja múltiplo de 400; 3) se o ano é múltiplo de 400, ele é obrigatoriamente um ano bissexto.

Exercício 3

Em cada item abaixo, fornecemos um vetor formado pela função `c()`. Perceba que em cada um desses vetores, valores de diferentes tipos são misturados. Como definimos na seção **Coerção no R**, quando dados de diferentes tipos são colocados dentro de um vetor atômico, o R automaticamente realiza um processo de coerção, ao converter todos esses dados para um único tipo. O seu objetivo nessa questão é simplesmente advinhar o tipo (`double`, `integer`, `logical`, ou `character`) de dado, para o qual esses dados serão convertidos. Caso esteja na dúvida, basta copiar e colar o código em seu console que você terá uma visão do resultado.

- A) `c(1.2, 2.4, "3.1", 1.9)`

- B) `integers <- 1:3`
`doubles <- c(2.23, 9.87, 3.2)`
`c(integers, doubles)`
- C) `c(1.56, 3L, 1L, 5L, 2.32, 9.87)`
- D) `c(TRUE, 1.5, FALSE)`
- E) `c("p", "b", "c", TRUE, 2L, 4.318)`

Exercício 4

Os próximos exercícios serão voltados para *subsetting*. Ao longo desses exercícios, estaremos utilizando o data.frame `flights`, que provém do pacote `nycflights13`. Por isso, lembre-se que para ter acesso a essa tabela, é necessário que você chame por esse pacote em sua sessão, com o comando `library()`. Caso você não tenha o pacote instalado em sua máquina, execute o comando `install.packages()` mostrado abaixo.

```
### Caso você não possua o pacote nycflights13
### instalado, execute o comando:

### install.packages("nycflights13")

library(nycflights13)

### Após o comando library() você
### terá acesso à tabela flights
flights
```

- A) Encontre todas as linhas da tabela `flights` em que `carrier` seja igual a "B6", e que `month` seja igual a 5.
- B) Todos os voês descritos na tabela `flights`, correspondem a voês que ocorreram no aeroporto de Nova York, ao longo do ano de 2013. A coluna `dep_delay` apresenta o tempo de atraso (em minutos) no momento de partida do aeroporto, e a coluna `arr_delay` apresenta o tempo de atraso (em minutos) no momento de chegada ao aeroporto. Tendo isso em mente, no ano de 2013, quantos voês obtiveram um tempo de atraso total acima do tempo médio de atraso?
- C) Selecione as linhas da tabela `flights` que se encaixam em pelo menos uma dessas duas condições: 1) possuem um `arr_delay` abaixo de 2 minutos, e que o aeroporto de destino (`dest`) seja "BOS"; 2) cujo horário de partida programado (`sched_dep_time`) seja de 6:00 (ou 600), e que o mês de ocorrência do voê seja igual a 1.

Capítulo 3

Importando e exportando dados com o R

3.1 Introdução e pré-requisitos

Em algum ponto, você vai trabalhar com os seus próprios dados no R e, para isso, você precisa obrigatoriamente importar esses dados para dentro do R. Neste capítulo, vamos aprender como utilizar as funções dos pacotes `readr`, `readxl` e `haven`, para ler e importar dados presentes em arquivos de texto (*plain text files* - `.txt` ou `.csv`), em planilhas do Excel (`.xlsx`) e em arquivos produzidos por programas estatísticos como o Stata (`.dta`), SPSS (`.sav`; `.zsav` e `.por`) e SAS (`.sas`).

Para que você tenha acesso as funções e possa acompanhar os exemplos desse capítulo você precisa chamar pelos pacotes `readr`, `readxl` e `haven`, através do comando `library()`. O pacote `readr` especificamente, está incluso dentro do `tidyverse` e, por isso, você também pode chamar por ele.

```
library(tidyverse)
```

```
library(readr)
library(readxl)
library(haven)
```

3.2 Fontes de dados

Os seus dados podem vir de diferentes tipos de fontes. Com isso, os métodos necessários para acessar e importar esses dados para o R, mudam. Em resumo, os seus dados podem provir de três tipos de fontes diferentes:

- 1) Arquivo estático salvo no disco rígido de seu computador.
- 2) Servidor local ou *online*.
- 3) Página da internet.

Nós normalmente transportamos os nossos dados através de um arquivo estático, que pode ser salvo em nosso computador. Por isso, ao importar os nossos dados para o R, vamos estar preocupados na grande maioria das vezes, em ler um arquivo que se encontra salvo em nosso computador. Por este motivo, os métodos que serão mostrados nesse capítulo, buscam ler e importar diferentes tipos de arquivos estáticos. Em um processo como esse, a localização desse arquivo no disco rígido é um fator importante. Além disso, diferentes tipos de arquivos são estruturados de maneiras distintas e, por essa razão, você vai precisar de uma função no R que seja capaz de ler esse tipo de arquivo, ou em outras palavras, que seja capaz de reconhecer a estrutura desse arquivo.

Por outro lado, se você está importando os seus dados a partir de um servidor, você muito provavelmente estará extraindo dados de um DBMS (*database management system*). Os sistemas DBMS mais famosos e mais utilizados no mundo, são os sistemas que utilizam a linguagem SQL (*Structured Query Language*). Para extrair um conjunto de dados desse tipo de fonte, você em geral necessita de uma chave, ou uma API (*Application Programming Interface*) que lhe garanta acesso ao servidor e, portanto, acesso aos dados que você deseja importar. Esse processo de importação não será tratado aqui, mas você vai precisar das funções disponíveis em pacotes como *jsonlite*, *odbc* e *DBI* para tal processo.

Além dessas alternativas, você pode estar interessado em coletar dados de uma página da internet. Não estou me referindo a um arquivo estático que esteja disponível para *download* através dessa página da internet, mas sim, de coletar o conteúdo dessa página, de coletar os dados que formam a própria página da internet em si. Esse tipo de coleta, e os métodos envolvidos nesse processo, são comumente chamados de *web scraping*, e hoje, representam uma área importante em análise de dados. Esse processo de importação também não será mostrado aqui, mas você pode consultar as funções dos pacotes *httr*, *xml2* e *rvest* para executar tal processo.

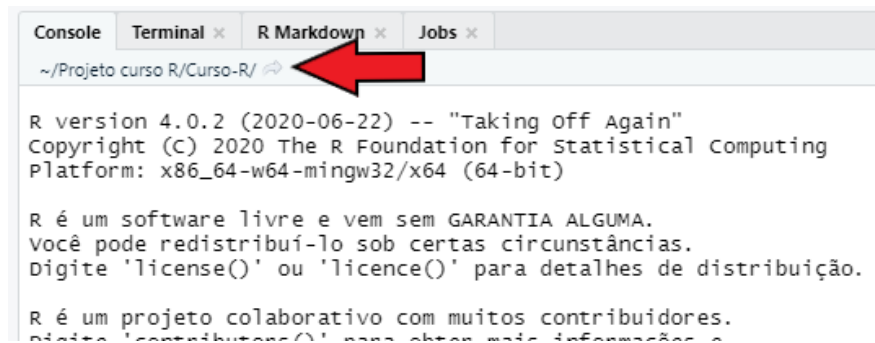
3.3 Diretório de trabalho

A linguagem R possui uma forte noção de diretórios de trabalho (WICKHAM; GROLEMUND, 2017, p. 113). O diretório de trabalho (*working directory*) é o local de seu computador onde o R vai procurar pelos arquivos que você demanda, e será onde o R vai guardar todos os arquivos que você pede a ele que salve.

Isso significa que em todas as ocasiões em que você estiver no R, ele estará trabalhando com alguma pasta específica de seu computador. No RStudio, você pode identificar o seu diretório de trabalho atual na parte esquerda e superior do console, logo abaixo do nome de sua guia (Console), como mostrado na figura 3.1. Repare abaixo, que no momento em que a foto presente na figura 3.1 foi tirada, eu estava trabalhando com uma pasta de meu computador chamada Curso-R, que por sua vez, se encontrava dentro de uma pasta chamada Projeto curso R.

Nós também podemos descobrir o diretório de trabalho atual em nossa sessão do R, através da função `getwd()`.

`getwd()`

Figura 3.1: Diretório de trabalho - Console RStudio

Fonte: Elaboração própria do autor.

```
## [1] "C:/Users/Pedro/Documents/Projeto curso R/Curso-R"
```

Dessa forma, supondo que o meu diretório de trabalho atual seja a pasta Curso-R, se eu pedir por algum arquivo chamado frase.txt, o R vai procurar por esse arquivo dentro dessa pasta Curso-R. Isso tem duas implicações muito importantes. Primeiro, o arquivo frase.txt deve estar dentro dessa pasta Curso-R, caso contrário o R não poderá encontrar o arquivo. Segundo, temos uma maneira muito simples e poderosa de acessarmos qualquer arquivo que esteja presente na pasta Curso-R, pois precisamos apenas do nome desse arquivo, como no exemplo abaixo.

```
read_lines("frase.txt")
```

```
## [1] "Aristóteles foi um filósofo da Grécia Antiga"
```

Um ponto muito importante é que a extensão do arquivo (que traduz o seu tipo) também faz parte do nome do arquivo. No exemplo acima, o arquivo se chama frase e possui a extensão .txt, logo, o nome do arquivo a ser fornecido ao R é frase.txt.

3.4 Definindo endereços do disco rígido no R

Portanto, o mecanismo de diretórios de trabalho apenas limita o escopo de busca do R. Dito de outra forma, ele define onde o R irá procurar pelos seus arquivos e, onde esses arquivos serão salvos através do R. Entretanto, isso não quer dizer que você não possa acessar arquivos que se encontram em outras áreas do seu computador. Porém, para acessarmos qualquer arquivo que esteja fora de seu diretório de trabalho atual, nós precisamos obrigatoriamente fornecer o endereço até esse arquivo para o R.

3.4.1 Cuidados ao definir endereços

Alguns cuidados no R são necessários ao definir um endereço até um arquivo. Primeiro, endereços de seu disco rígido devem sempre ser fornecidos como textos (*strings*), por isso, lembre-se de contornar o seu endereço com aspas duplas ou simples no R. Segundo, o Windows utiliza por padrão a barra inclinada à esquerda (\) para separar cada diretório presente no caminho até um certo arquivo. Todavia, a barra inclinada à esquerda possui um significado especial para o R.

Abordando especificamente o segundo ponto, você tem duas alternativas para contornar as particularidades das barras inclinadas utilizadas nos endereços de seus arquivos: 1) utilizar o estilo dos sistemas Mac e Linux, que utilizam a barra inclinada à direita (/) para separar os diretórios; 2) ou contornar o comportamento especial de uma barra inclinada à esquerda, com duas barras inclinadas à esquerda (\\). Ou seja, é como se essas duas barras \\ significassem apenas uma barra \ para o R. Eu particularmente prefiro utilizar o estilo dos sistemas Mac e Linux para resolver esse problema, pois ele incorre em um trabalho menor de digitação.

Por exemplo, eu possuo um arquivo chamado `livros.txt` localizado dentro da pasta `Lista de compras`, que por sua vez, se encontra dentro da minha pasta de Documentos do Windows. Segundo o padrão do Windows, o endereço até esse arquivo seria: "C:\Users\Pedro\Documents\Lista de compras\livros.txt". Porém, levando-se em conta os pontos que acabamos de abordar, nós poderíamos fornecer um dos dois endereços abaixo para me referir a este arquivo no R:

```
livros <- read_csv("C:\\Users\\Pedro\\Documents\\Lista de compras\\livros.txt")
```

```
livros <- read_csv("C:/Users/Pedro/Documents/Lista de compras/livros.txt")
```

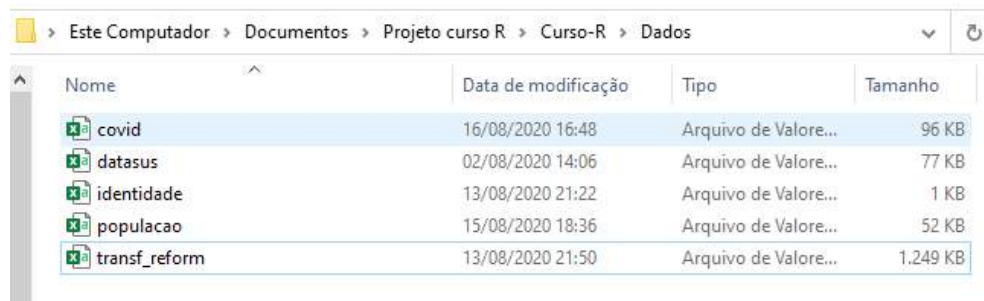
```
livros
```

```
## # A tibble: 4 x 3
##   Título                                Autor                                Preço
##   <chr>                                <chr>                                <dbl>
## 1 O Hobbit                            J. R. R. Tolkien                    40.7
## 2 Matemática para Economistas          Carl P. Simon e Lawrence Blume      140.
## 3 Microeconomia: uma abordagem moderna Hal R. Varian                      142.
## 4 A Luneta Âmbar                      Philip Pullman                      42.9
```

3.4.2 Endereços relativos e absolutos

A depender do seu diretório de trabalho atual, e de onde o seu arquivo de interesse se encontra, você pode utilizar dois estilos diferentes de endereços (relativo e absoluto) para se referir a um dado arquivo. Vamos utilizar como exemplo, o conjunto de arquivos mostrados na figura 3.2 que se encontram dentro de uma pasta chamada `Dados`.

Caso o seu diretório de trabalho atual fosse, por exemplo, a pasta `Projeto curso R`, você poderia fornecer um endereço relativo para qualquer um desses arquivos presentes na pasta `Dados`. Pois a

Figura 3.2: Exemplo de arquivos


The screenshot shows a Windows File Explorer window with the address bar set to 'Este Computador > Documentos > Projeto curso R > Curso-R > Dados'. The main pane displays a list of five CSV files with columns for Name, Last Modified, Type, and Size.

Nome	Data de modificação	Tipo	Tamanho
covid	16/08/2020 16:48	Arquivo de Valore...	96 KB
datasus	02/08/2020 14:06	Arquivo de Valore...	77 KB
identidade	13/08/2020 21:22	Arquivo de Valore...	1 KB
populacao	15/08/2020 18:36	Arquivo de Valore...	52 KB
transf_reform	13/08/2020 21:50	Arquivo de Valore...	1.249 KB

Fonte: Elaboração própria do autor.

pasta Dados se encontra dentro da pasta Projeto curso R. Em outras palavras, a pasta Dados é uma subpasta da pasta Projeto curso R.

Logo, um endereço relativo possui como ponto inicial, o seu diretório de trabalho atual. Por isso, você pode acessar qualquer arquivo que esteja dentro de seu diretório de trabalho, ou dentro de alguma de suas subpastas, através de um endereço relativo. No caso dos arquivos da pasta Dados, nós poderíamos fornecer o endereço "Curso-R/Dados/" para chegarmos a pasta Dados. Em seguida, precisaríamos apenas acrescentar o nome do arquivo de nosso desejo. Por exemplo, se fôssemos ler o arquivo de nome covid.csv, o endereço resultante seria "Curso-R/Dados/covid.csv".

Por outro lado, se o seu diretório de trabalho atual for uma pasta posterior à pasta Dados (ou seja, uma subpasta da pasta Dados), e você quiser acessar um dos arquivos da pasta Dados, você terá que fornecer um endereço absoluto até o arquivo em questão. Um endereço absoluto é um endereço que parte desde o disco rígido de seu computador até o arquivo de interesse. Por isso, um endereço absoluto sempre aponta para o mesmo local de seu computador, independente de qual seja o seu diretório de trabalho atual.

Para coletarmos o endereço absoluto de um arquivo no Windows, podemos clicar com o botão direito do mouse sobre o arquivo de interesse, e selecionar a opção Propriedades. Uma caixa vai abrir em sua tela, contendo diversas informações sobre o arquivo em questão. Logo a sua frente, temos a seção chamada Local na parte inicial dessa caixa, onde podemos encontrar o endereço absoluto até a pasta onde o seu arquivo de interesse está localizado.

Logo, se eu utilizasse esse recurso sobre um dos arquivos mostrados na figura 3.2, eu encontraria o seguinte endereço nessa seção Local: "C:\Users\Pedro\Documents\Projeto curso R\Curso-R\Dados". Com esse endereço, precisamos apenas adicionar o nome do arquivo desejado, e ajustar as barras inclinadas à esquerda de acordo com as alternativas apresentadas na seção anterior. Por exemplo, se o nosso arquivo de interesse fosse o covid.csv, o endereço absoluto a ser fornecido ao R seria: "C:/Users/Pedro/Documents/Projeto curso R/Curso-R/Dados/covid.csv".

Segundo Wickham e Grolemund (2017), é recomendável que você evite endereços absolutos, especialmente se você trabalha em conjunto. Pois é muito provável que os computadores de seus

parceiros de trabalho não possuem exatamente a mesma estrutura de diretórios que o seu computador. Por isso, o ideal é que você sempre organize todos os arquivos referentes a um certo projeto ou a uma certa análise, dentro de uma pasta específica de seu computador. Dessa forma, você pode tornar essa pasta específica o seu diretório de trabalho no R, e a partir daí, fornecer endereços relativos até cada arquivo.

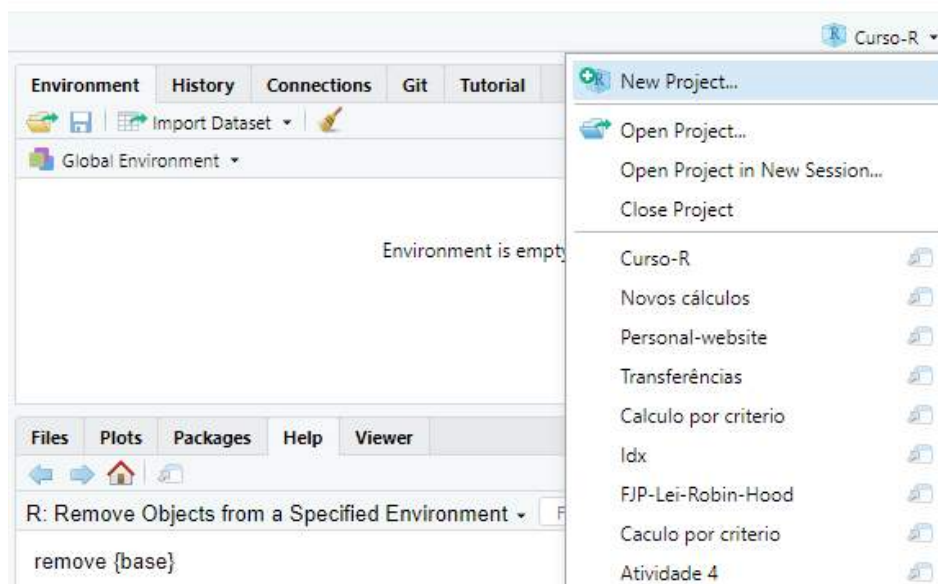
3.5 Plataforma de Projetos do RStudio

No R, você pode configurar o seu diretório de trabalho atual, através da função `setwd()`. Basta fornecer o endereço absoluto até a pasta com a qual você deseja trabalhar. Veja o exemplo abaixo, em que eu escolho a pasta de Documentos do Windows como o meu diretório de trabalho:

```
setwd("C:/Users/Pedro/Documents")
```

Porém, esse não é um método recomendado de se configurar o seu diretório de trabalho, especialmente porque nós precisamos realizar essa configuração toda vez em que acessamos o R, sendo algo contraproducente. Por isso, [Wickham e Grolemund \(2017\)](#) caracterizam a plataforma de Projetos do RStudio, como uma forma mais adequada e eficiente de realizarmos essa configuração.

Figura 3.3: Plataforma de Projetos do RStudio - Parte 1



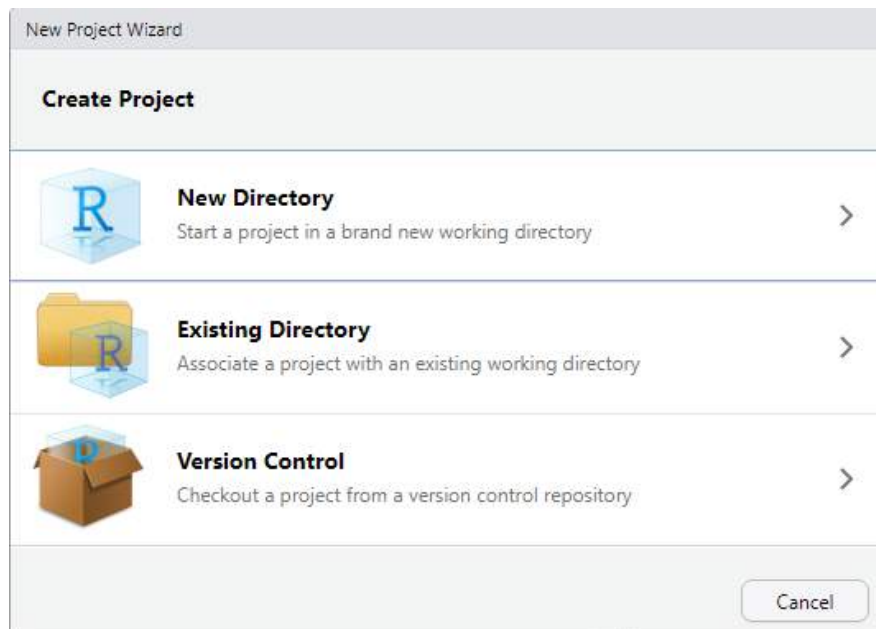
Fonte: Elaboração própria do autor.

Ao criar um projeto no RStudio, você está apenas criando um arquivo com o nome desse projeto e que possui uma extensão `.Rproj`. Esse arquivo `.Rproj` funciona como um link até a pasta onde você o guardou. Dessa forma, ao acessarmos esse projeto no RStudio, o seu console já vai estar

trabalhando com a pasta onde o arquivo `.Rproj` foi salvo. Em termos técnicos, toda vez que você acessar esse projeto, o RStudio vai automaticamente configurar essa pasta como o seu diretório de trabalho atual do R.

Para criarmos um projeto no RStudio, você pode acessar um pequeno menu localizado na parte superior e direita de sua tela, mostrado na figura 3.3. Ao selecionar a opção `New Project...`, o seu RStudio vai abrir uma aba que está exposta na figura 3.4. Nessa aba, você vai selecionar como deseja criar o novo arquivo `.Rproj`. Caso você já tenha organizado todos os arquivos de seu projeto em uma pasta específica, você pode selecionar a opção `Existing Directory` para salvar o arquivo `.Rproj` nessa pasta já existente. Por outro lado, caso você esteja iniciando a sua análise do zero, você pode selecionar a opção `New Directory` para criar um novo diretório em seu computador, onde você vai guardar todos os arquivos referentes ao seu projeto.

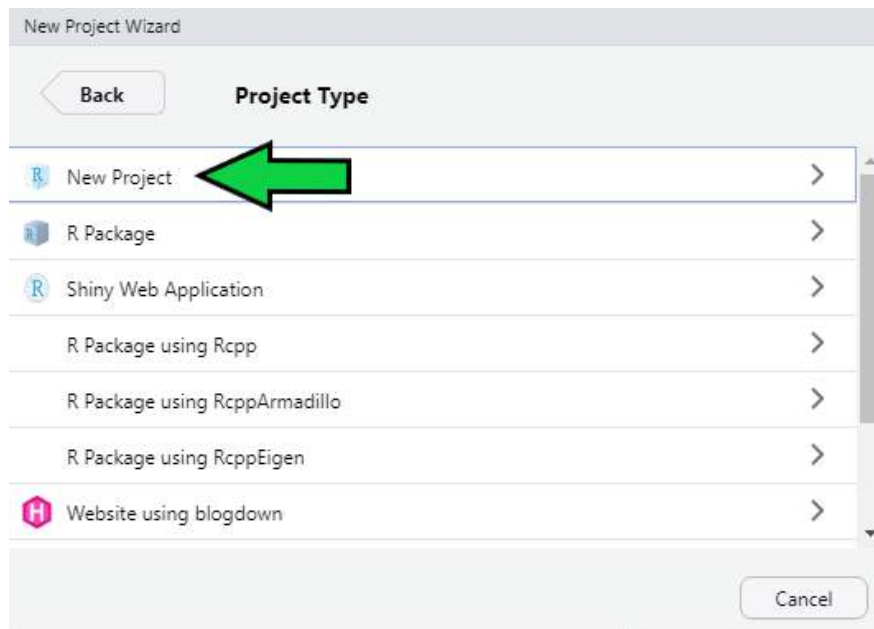
Figura 3.4: Plataforma de Projetos do RStudio - Parte 2



Fonte: Elaboração própria do autor.

Ao selecionar uma dessas opções, o RStudio também vai lhe questionar sobre o tipo desse projeto, ou dito de outra maneira, qual o tipo de produto que você busca gerar com esse projeto, através da aba mostrada na figura 3.5. Ou seja, se você está planejando construir um novo pacote para o R, é interessante que você selecione a segunda opção (`R Package`) dessa aba. Pois assim, o próprio RStudio vai automaticamente criar para você, os principais arquivos que um pacote do R precisa ter. Em geral, você vai selecionar a primeira opção (`New Project`) para criar um projeto padrão.

No exemplo apresentado pela figura 3.6, eu estou criando um projeto padrão chamado `projeto_mortalidade` na pasta `Desktop` (que corresponde a área de trabalho) de meu computador. Com isso, uma nova pasta chamada `projeto_mortalidade` será criada, e sempre que eu acessar

Figura 3.5: Plataforma de Projetos do RStudio - Parte 3

Fonte: Elaboração própria do autor.

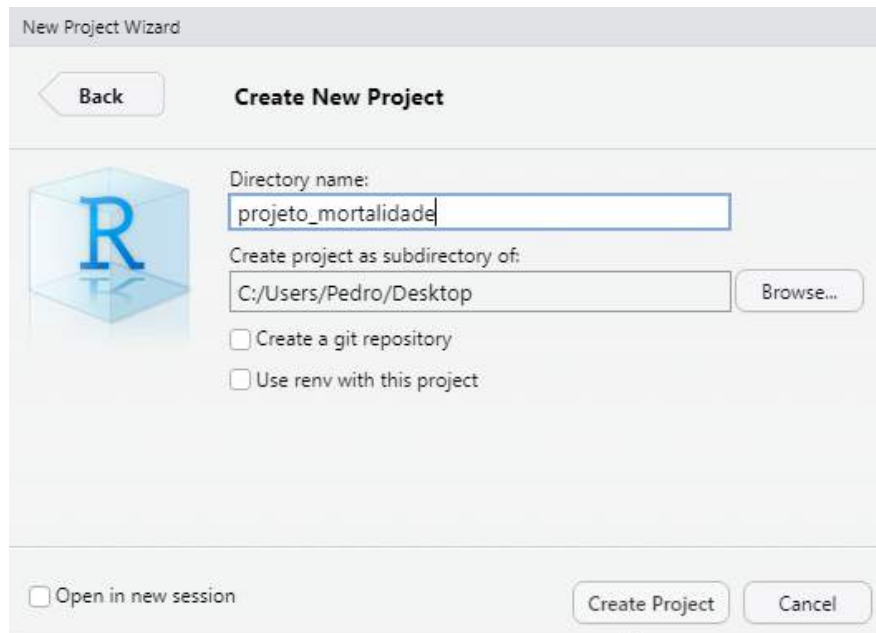
novamente o projeto `projeto_mortalidade` no RStudio, através do pequeno menu mostrado na figura 3.3, o RStudio vai automaticamente configurar a pasta `projeto_mortalidade` como o diretório de trabalho atual do R.

3.6 Importando arquivos de texto com readr

Arquivos de texto, também conhecidos como *plain text files*, ou *flat files*, estão entre os formatos de arquivo mais utilizados em todo o mundo para transportar e armazenar dados. Por isso é muito importante que você conheça esses arquivos e saiba reconhecê-los.

Um arquivo de texto, normalmente assume a extensão `.txt`, e contém apenas cadeias de textos ou cadeias de valores numéricos que são organizados em linhas. Apesar de simples, os dados armazenados podem ser organizados de diferentes formas em cada linha do arquivo. Por essa razão, um arquivo de texto pode assumir diferentes extensões que identificam o tipo de arquivo de texto ao qual ele pertence.

Em outras palavras, nós possuímos diferentes tipos de arquivos de texto, e a diferença básica entre eles, está na forma como os valores são organizados em cada linha do arquivo. Um dos tipos de arquivo de texto mais famosos é o arquivo CSV (*comma separated file*), que utiliza vírgulas (ou pontos e vírgulas como é o caso brasileiro) para separar os valores de diferentes colunas em cada linha do arquivo. Por isso, não basta que você identifique se o seu arquivo de interesse é um arquivo

Figura 3.6: Plataforma de Projetos do RStudio - Parte 4

Fonte: Elaboração própria do autor.

de texto, pois você também precisa identificar o **tipo** de arquivo de texto no qual ele se encaixa.

Para importarmos os dados presentes nesses arquivos, vamos utilizar as funções do pacote `readr`, que oferece um conjunto de funções especializadas em arquivos de texto. Logo abaixo, temos uma lista que associa os respectivos tipos de arquivos de texto a cada uma das funções desse pacote.

- 1) `read_delim()`: essa é uma função geral, que é capaz de ler qualquer tipo de arquivo de texto em que os valores estão delimitados por algum caractere especial.
- 2) `read_csv2()`: lê arquivos CSV (*comma separated file*) que seguem o padrão adotado por alguns países europeus. Arquivos `.txt` ou `.csv`, em que os valores são separados por ponto e vírgula (;).
- 3) `read_csv()`: lê arquivos CSV (*comma separated file*) que seguem o padrão americano. Arquivos `.txt` ou `.csv` onde os valores são separados por vírgula (,).
- 4) `read_tsv()`: lê arquivos TSV (*tab separated values*). Arquivos `.txt` ou `.tsv` onde os valores são separados por tabulação (`\t`).
- 5) `read_fwf()`: lê arquivos FWF (*fixed width file*). Arquivos `.txt` ou `.fwf` onde cada coluna do arquivo possui uma largura fixa de valores.

Perceba que o nome de todas as funções acima seguem o padrão `read_*`, onde a palavra presente no ponto `*` corresponde a extensão que identifica o tipo de arquivo no qual a função é especializada.

Nós sempre iniciamos qualquer uma das funções acima, pelo endereço até o arquivo que desejamos ler. Como exemplo inicial, eu possuo um arquivo CSV chamado `Censo_2010.csv`, que se encontra dentro da pasta 6 - Importacao.

```
library(readr)

Censo_2010 <- read_csv2("Parte 1/6 - Importacao/Censo_2010.csv")

## -- Column specification -----
## cols(
##   `Região metropolitana` = col_character(),
##   `População residente` = col_double(),
##   `População em área urbana` = col_double(),
##   `População em área não urbanizada` = col_double(),
##   `População em área isolada` = col_double(),
##   `Área rural` = col_double(),
##   `Aglomerado urbano` = col_double(),
##   Povoado = col_double(),
##   Núcleo = col_double(),
##   `Outros aglomerados` = col_double(),
##   `Código unidade` = col_double()
## )
```

Perceba também no exemplo acima, que eu salvo o resultado da função `read_csv2()` em um objeto chamado `Censo_2010`. Isso é muito importante! Lembre-se sempre de salvar o resultado das funções `read_*` em algum objeto. Pois a função `read_csv2()` busca apenas ler o arquivo `Censo_2010.csv` e encaixar o seu conteúdo em uma tabela (ou um `data.frame`) do R. Ou seja, em nenhum momento, a função `read_csv2()` se preocupa em salvar os dados que ela coletou do arquivo `Censo_2010.csv`, em algum lugar que podemos acessar futuramente. É por essa razão, que eu salvo a tabela gerada pela função `read_csv2()` em um objeto. Pois dessa forma, eu posso acessar novamente os dados que coletamos do arquivo `Censo_2010.csv`, através do objeto `Censo_2010`.

```
Censo_2010
```

```
## # A tibble: 2,013 x 11
##   `Região metropol~` `População resid~` `População em ár~` `População em áre~
##   <chr>                <dbl>                <dbl>                <dbl>
## 1 Manaus AM            2106322            1972885            3011
## 2 Homens                1036676            964041            2018
## 3 Mulheres             1069646            1008844            993
## 4 Careiro da Várzea    23930              1000              NA
## 5 Homens                12688              481              NA
## 6 Mulheres             11242              519              NA
## 7 Iranduba             40781             28979             NA
## 8 Homens                20996             14662             NA
```

```
## 9 Mulheres 19785 14317 NA
## 10 Itacoatiara 86839 57863 294
## # ... with 2,003 more rows, and 7 more variables:
## #   População em área isolada <dbl>, Área rural <dbl>,
## #   Aglomerado urbano <dbl>, Povoado <dbl>, Núcleo <dbl>,
## #   Outros aglomerados <dbl>, Código unidade <dbl>
```

Mesmo que o arquivo `Censo_2010.csv` seja claramente um arquivo CSV, nós precisamos identificar qual o padrão que ele está adotando. Nos EUA, um arquivo CSV utiliza vírgulas (,) para separar os valores de cada coluna. Porém, pelo fato de nós, brasileiros, usarmos a vírgula para delimitar casas decimais em números reais, nós empregamos o padrão de um arquivo CSV adotado por alguns países europeus, que utilizam o ponto e vírgula (;) como separador. Logo abaixo, temos as linhas iniciais do arquivo `Censo_2010.csv` e, podemos rapidamente identificar que esse arquivo utiliza o padrão europeu. É por este motivo que eu utilizo a função `read_csv2()`, e não a função `read_csv()` para ler o arquivo.

```
Manaus AM;2106322;1972885;3011;;108160;;22266;;;30
Homens;1036676;964041;2018;;59024;;11593;;;30
Mulheres;1069646;1008844;993;;49136;;10673;;;30
Careiro da Várzea;23930;1000;;21089;;1841;;1301159
Homens;12688;481;;11281;;926;;1301159
Mulheres;11242;519;;9808;;915;;1301159
```

Apesar de ser esse o padrão adotado por nós brasileiros, você enfrentará ocasiões em que o seu arquivo de texto possui separadores diferentes do esperado. Por exemplo, talvez os seus dados sejam separados por cifrões (\$).

```
t <- "Ano$Código$Dia$Valor
2020$P.A22$01$4230.45
2020$B.34$02$1250.28
2020$S.T4$03$3510.90"
```

```
writeLines(t)
```

```
## Ano$Código$Dia$Valor
## 2020$P.A22$01$4230.45
## 2020$B.34$02$1250.28
## 2020$S.T4$03$3510.90
```

Em casos como esse, você será obrigado a definir explicitamente o separador utilizado no arquivo. Para isso, você pode utilizar a função `read_delim()`, que possui o argumento `delim`, onde podemos determinar o caractere que delimita as colunas no arquivo.

```
read_delim(t, delim = "$")
```



```
## # A tibble: 3 x 4
##   Ano Código Dia  Valor
##   <dbl> <chr> <chr> <dbl>
## 1  2020 P.A22  01    4230.
## 2  2020 B.34   02    1250.
## 3  2020 S.T4   03    3511.
```

Como um outro exemplo, arquivos TSV são simplificadaamente um arquivo CSV que utiliza um caractere especial de tabulação como separador, representado pelos caracteres `\t`. Ou seja, nós podemos recriar a função `read_tsv()` através da função `read_delim()`, ao configurarmos o argumento `delim`, como no exemplo abaixo.

```
t <- "Ano\tCódigo\tDia\tValor
2020\tP.A22\t01\t4.230,45
2020\tB.34\t02\t1.250,28
2020\tS.T4\t03\t3.510,90"
```

```
writeLines(t)
```

```
## Ano  Código  Dia Valor
## 2020 P.A22   01 4.230,45
## 2020 B.34    02 1.250,28
## 2020 S.T4    03 3.510,90
```

```
read_delim(t, delim = "\t")
```

```
## # A tibble: 3 x 4
##   Ano Código Dia  Valor
##   <dbl> <chr> <chr> <dbl>
## 1  2020 P.A22  01    4.23
## 2  2020 B.34   02    1.25
## 3  2020 S.T4   03    3.51
```

3.6.1 Definindo os tipos de dados em cada coluna

Caso nós não informarmos em qualquer uma das funções `read_*`, qual o tipo de dado contido em cada coluna de nosso arquivo de texto, essas funções vão por padrão, ler as 1000 primeiras linhas de seu arquivo, e com base nessas 1000 linhas, vão tentar adivinhar qual o tipo de dado contido em cada coluna. Após esse processo, a função `read_*` vai ler as linhas restantes do arquivo, se baseando nos tipos identificados pela função.

Tendo isso em mente, todas as funções `read_*` sempre nos fornecem uma pequena descrição, contendo a especificação de cada coluna (Column specification). Essa descrição está nos informando justamente qual foi o “chute” da função, ou qual o tipo de dado que a função utilizou em cada coluna. Veja no exemplo abaixo, que a função `read_csv()` interpretou que as colunas Título e Autor

continham dados textuais e, por isso, utilizou colunas do tipo `character` (`col_character()`) para guardar esses dados. Por outro lado, a função percebeu que a coluna `Preço` continha dados numéricos e, por essa razão, preferiu utilizar uma coluna do tipo `double` (`col_double()`) para alocar esses dados na tabela.

```
livros <- read_csv("C:/Users/Pedro/Documents/Lista de compras/livros.txt")

-- Column specification -----
cols(
  Título = col_character(),
  Autor = col_character(),
  Preço = col_double()
)
```

Isso é uma característica importante e útil das funções `read_*`, pois podemos contar com esse sistema para definir os tipos de cada coluna do arquivo. Porém, esse é um sistema que se torna cada vez mais frágil a medida em que o tamanho de nosso arquivo aumenta. Pois essas 1000 primeiras linhas começam a representar uma parte cada vez menor do arquivo e, portanto, as suas chances de demonstrarem fielmente os tipos de dados presentes em todo arquivo, ficam cada vez menores.

Por isso, é provável que em algum momento, você terá de contornar esse comportamento, e definir explicitamente os tipos de dados contidos em cada coluna por meio do argumento `col_types` de qualquer função `read_*`.

Para construirmos essa definição, nós utilizamos a função `cols()` e suas variantes `col_*`. Dentro da função `cols()`, precisamos igualar o nome da coluna presente no arquivo de texto à função `col_*` que corresponde ao tipo de dado desejado. No exemplo abaixo, ao igualar as colunas `year`, `month` e `day` à função `col_integer()`, eu estou definindo que essas colunas devem ser interpretadas como colunas do tipo `integer`. Enquanto isso, ao igualar as colunas `carrier` e `tailnum` à função `col_character()`, eu estou requisitando que essas colunas sejam lidas como colunas do tipo `character`.

Por outro lado, a função `cols()` nos oferece um atalho chamado `.default`. Mediante esse atalho, podemos nos referir a todas as colunas do arquivo de uma vez. Por isso, no exemplo abaixo, ao igualar esse atalho à função `col_double()`, eu estou dizendo à função `cols()`, que qualquer outra coluna do arquivo que não tenha sido definida explicitamente na função `cols()`, deve ser interpretada como uma coluna do tipo `double`. Por este motivo, as colunas `dep_time` e `dep_delay` (e várias outras), que não foram configuradas explicitamente na função `cols()`, acabaram sendo interpretadas como colunas do tipo `double`.

```
tipos_col <- cols(
  .default = col_double(),
  year = col_integer(),
  month = col_integer(),
  day = col_integer(),
  carrier = col_character(),
```

```

  tailnum = col_character(),
  origin = col_character(),
  dest = col_character(),
  time_hour = col_datetime(format = "")
)

```

```

flights <- read_csv2(
  "flights.csv",
  col_types = tipos_col
)

```

```
flights
```

```

## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <dbl>         <dbl>       <dbl>   <dbl>
## 1  2013     1     1     517             515         2     830
## 2  2013     1     1     533             529         4     850
## 3  2013     1     1     542             540         2     923
## 4  2013     1     1     544             545        -1    1004
## 5  2013     1     1     554             600        -6     812
## 6  2013     1     1     554             558        -4     740
## 7  2013     1     1     555             600        -5     913
## 8  2013     1     1     557             600        -3     709
## 9  2013     1     1     557             600        -3     838
## 10 2013     1     1     558             600        -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <dbl>,
## #   arr_delay <dbl>, carrier <chr>, flight <dbl>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>

```

3.6.2 Compreendendo o argumento locale

O argumento `locale` está presente em todas as funções `read_*`, e é responsável por definir as especificações do arquivo de texto que mudam de país para país. No Brasil, por exemplo, datas são definidas no formato “Dia/Mês/Ano”, enquanto nos EUA, datas se encontram no formato “Ano-Mês-Dia”. No Brasil, utilizamos vírgulas para separar a parte decimal de um número, enquanto nos EUA, essa separação é definida por um ponto final. Uma diferença ainda mais importante, se encontra no sistema de *encoding* adotado, que varia de maneira muito violenta ao longo dos países.

O R, é uma linguagem centrada nos padrões americanos, por isso, sempre que você estiver tentando ler algum arquivo de texto que não se encaixa de alguma forma neste padrão, você terá que ajustar o `locale` da função `read_*` que você está utilizando. Algumas funções já preveem e adotam essas diferenças, um exemplo disso, é a função `read_csv2()`, que é na verdade um atalho para o padrão adotado por nós brasileiros, e por alguns países europeus.

Como exemplo inicial, vamos tentar ler o arquivo `pib_per_capita.csv`, que novamente se encontra dentro da pasta 6 - Importacao. Dessa vez, vamos utilizar a função geral do pacote, a `read_delim()`. Lembre-se que nessa função, você deve sempre indicar qual o caractere separador do arquivo, através do argumento `delim`.

```
pib <- read_delim("Parte 1/6 - Importacao/pib_per_capita.csv", delim = ";")
```

```
pib
```

```
## # A tibble: 853 x 7
##   IBGE2    IBGE `Munic\`xedpio`      `Popula\`xe7\`xe3o`  Ano PIB
##   <dbl>  <dbl> <chr>                        <dbl> <dbl> <chr>
## 1    10 310010 "Abadia dos Dourados"          6972  2017 33.389.769,00
## 2    20 310020 "Abaet\`xe9"                23223  2017 96.201.158,00
## 3    30 310030 "Abre Campo"                13465  2017 29.149.429,00
## 4    40 310040 "Acaiaca"                3994  2017 2.521.892,00
## 5    50 310050 "A\`xe7ucena"              9575  2017 15.250.077,00
## 6    60 310060 "\`xc1gua Boa"          13600  2017 29.988.906,00
## 7    70 310070 "\`xc1gua Comprida"        2005  2017 74.771.408,00
## 8    80 310080 "Aguanil"                4448  2017 15.444.038,00
## 9    90 310090 "\`xc1guas Formosas"      19166  2017 11.236.696,00
## 10  100 310100 "\`xc1guas Vermelhas"        13477  2017 48.088.397,00
## # ... with 843 more rows, and 1 more variable: PIB per capita <chr>
```

Algo deu errado durante a importação, pois as colunas PIB e PIB per capita foram importadas como colunas de texto (character), sendo que elas são claramente numéricas. Em momentos como esse, é interessante que você consulte as primeiras linhas do arquivo, para compreender melhor a sua estrutura e identificar o que deu errado. Por isso, temos logo abaixo, as três primeiras linhas do arquivo `pib_per_capita.csv`. Perceba que os dois últimos valores em cada linha, representam os dados das colunas PIB e PIB per capita. Ao olharmos, por exemplo, para o número 33.389.769,00 nós podemos identificar qual o problema que está ocorrendo em nossa importação.

```
10;310010;Abadia dos Dourados;6972;2017;33.389.769,00;4.789,12
20;310020;Abaeté;23223;2017;96.201.158,00;4.142,49
30;310030;Abre Campo;13465;2017;29.149.429,00;2.164,83
```

O motivo para tal conflito, se encontra justamente no uso do ponto final como separador de milhares, e da vírgula para marcar a parte decimal dos números dispostos nas colunas PIB e PIB_per_capita. Ou seja, como não informamos nada sobre as particularidades do arquivo, a função `read_delim()` está imaginando que o arquivo `pib_per_capita.csv` se encontra no padrão americano. Por isso, nós precisamos fornecer essas informações à função `read_delim()` para que esse problema seja corrigido, através do argumento `locale`.

Na verdade, tais informações são fornecidas através da função `locale()`, como no exemplo abaixo. No nosso caso, precisamos ajustar o caractere responsável por separar os milhares, que corresponde

ao argumento `grouping_mark`, e o caractere que defini a parte decimal dos nossos números, que corresponde ao argumento `decimal_mark` da função `locale()`. Perceba no exemplo abaixo, que ao provermos essas informações à função `read_delim()` através da função `locale()`, as colunas PIB e PIB per capita foram corretamente interpretadas como colunas numéricas (`double`).

```
pib <- read_delim(
  "Parte 1/6 - Importacao/pib_per_capita.csv",
  delim = ";",
  locale = locale(decimal_mark = ",", grouping_mark = ".")
)
```

```
pib
```

```
## # A tibble: 853 x 7
##   IBGE2   IBGE `Munic\xedpio`      `Popula\xe7\xe3o`  Ano    PIB
##   <dbl> <dbl> <chr>                <dbl> <dbl>   <dbl>
## 1    10 310010 "Abadia dos Dourados"      6972  2017 33389769
## 2    20 310020 "Abaet\xe9"              23223 2017 96201158
## 3    30 310030 "Abre Campo"              13465 2017 29149429
## 4    40 310040 "Acaiaca"                3994  2017  2521892
## 5    50 310050 "A\xe7ucena"              9575  2017 15250077
## 6    60 310060 "\xc1gua Boa"        13600 2017 29988906
## 7    70 310070 "\xc1gua Comprida"    2005  2017 74771408
## 8    80 310080 "Aguanil"              4448  2017 15444038
## 9    90 310090 "\xc1guas Formosas"   19166 2017 11236696
## 10  100 310100 "\xc1guas Vermelhas"    13477 2017 48088397
## # ... with 843 more rows, and 1 more variable: PIB per capita <dbl>
```

Apesar de resolvermos o problema gerado anteriormente nas colunas PIB e PIB per capita, ainda há algo que precisamos corrigir nessa importação. O problema remanescente, se encontra em colunas textuais e no título de algumas colunas. Perceba que alguns desses textos (especialmente em letras acentuadas) estão esquisitos. Por exemplo, a coluna que deveria se chamar Município está denominada como `Munic\xedpio`.

Esse é um típico problema de *encoding*, onde a função `read_delim()` imagina que o arquivo `pib_per_capita.csv` se encontra em um sistema de *encoding* específico, quando na verdade, ele se encontra em um outro sistema. Ou seja, tudo o que precisamos fazer, é informar qual o sistema correto de leitura do arquivo à função `read_delim()`. Por padrão, todas as funções do pacote `readr` vão pressupor que os seus arquivos se encontram no sistema UTF-8 de *encoding*. Porém, a maioria dos computadores brasileiros utilizam um outro sistema, sendo ele, o sistema ISO-8859-1, que também é conhecido por `Latin1`.

Nas funções do pacote `readr`, nós podemos definir o *encoding* de leitura, através do argumento `encoding` presente na função `locale()`. Nesse argumento, você pode fornecer tanto o nome oficial

do sistema (ISO-8859-1) quanto o seu apelido (Latin1). Repare no exemplo abaixo, que ao definirmos o *encoding* correto de leitura, os problemas em elementos textuais foram resolvidos. Para ter uma melhor compreensão desse problema, por favor leia a seção *Encoding de caracteres*.

```
pib <- read_delim(
  "Parte 1/6 - Importacao/pib_per_capita.csv",
  delim = ";",
  locale = locale(
    decimal_mark = ",",
    grouping_mark = ".",
    encoding = "Latin1"
  )
)
```

```
pib
```

```
## # A tibble: 853 x 7
##   IBGE2   IBGE Município      População  Ano    PIB `PIB per capita`
##   <dbl> <dbl> <chr>      <dbl> <dbl>  <dbl>      <dbl>
## 1    10 310010 Abadia dos Dourad~    6972 2017  3.34e7      4789.
## 2    20 310020 Abaeté          23223 2017  9.62e7      4142.
## 3    30 310030 Abre Campo      13465 2017  2.91e7      2165.
## 4    40 310040 Acaiaca         3994 2017  2.52e6       631.
## 5    50 310050 Açucena         9575 2017  1.53e7      1593.
## 6    60 310060 Água Boa       13600 2017  3.00e7      2205.
## 7    70 310070 Água Comprida    2005 2017  7.48e7     37292.
## 8    80 310080 Aguanil         4448 2017  1.54e7      3472.
## 9    90 310090 Águas Formosas  19166 2017  1.12e7       586.
## 10  100 310100 Águas Vermelhas  13477 2017  4.81e7      3568.
## # ... with 843 more rows
```

3.6.3 Outras configurações envolvendo linhas e colunas

Nessa seção, vamos utilizar como exemplo base, o arquivo CSV que forma o objeto `t` abaixo. Perceba que esse arquivo utiliza pontos e vírgulas como separador, e que ele não possui cabeçalho aparente. Ou seja, aparentemente os nomes das colunas não estão definidas no arquivo.

```
t <- "2020;P.A22;01;4230.45
2020;B.34;02;1250.28
2020;S.T4;03;3510.90
2020;B.35;04;1200.25
2020;F.J4;05;1542.20
2020;A.12;06;9854.09
2020;B.Q2;07;7654.10
2020;G.T4;08;4328.36"
```

```
2020;E.7A;09;2310.25"
```

```
read_delim(t, delim = ";")
```

```
## # A tibble: 8 x 4
##   `2020` P.A22 `01` `4230.45`
##   <dbl> <chr> <chr>    <dbl>
## 1   2020 B.34  02      1250.
## 2   2020 S.T4  03      3511.
## 3   2020 B.35  04      1200.
## 4   2020 F.J4  05      1542.
## 5   2020 A.12  06      9854.
## 6   2020 B.Q2  07      7654.
## 7   2020 G.T4  08      4328.
## 8   2020 E.7A  09      2310.
```

Por padrão, as funções `read_*` utilizam a primeira linha do arquivo para construir o nome de cada coluna presente. Mas se você deseja evitar esse comportamento, você pode configurar o argumento `col_names` para `FALSE`. Dessa forma, a função `read_*` vai gerar nomes genéricos para cada coluna. Uma outra alternativa é fornecer um vetor ao argumento `col_names`, contendo os nomes de cada coluna na ordem em que elas aparecem no arquivo, como no exemplo abaixo.

```
col <- c("Ano", "Código", "Dia", "Valor")
```

```
read_delim(t, delim = ";", col_names = col)
```

```
## # A tibble: 9 x 4
##   Ano Código Dia  Valor
##   <dbl> <chr> <chr> <dbl>
## 1  2020 P.A22 01   4230.
## 2  2020 B.34 02   1250.
## 3  2020 S.T4 03   3511.
## 4  2020 B.35 04   1200.
## 5  2020 F.J4 05   1542.
## 6  2020 A.12 06   9854.
## 7  2020 B.Q2 07   7654.
## 8  2020 G.T4 08   4328.
## 9  2020 E.7A 09   2310.
```

Além disso, as funções `read_*` nos permite determinar o número máximo de linhas que desejamos ler de um arquivo, através do argumento `n_max`. Logo, mesmo que um arquivo de texto qualquer possua 500 mil linhas, nós podemos ler apenas as 10 primeiras linhas desse arquivo, ao configurarmos esse argumento. No exemplo abaixo, eu estou lendo apenas as 5 primeiras linhas do arquivo `t`.

```
read_delim(t, delim = ";", n_max = 5, col_names = col)
```

```
## # A tibble: 5 x 4
##   Ano Código Dia  Valor
##   <dbl> <chr> <chr> <dbl>
## 1  2020 P.A22  01    4230.
## 2  2020 B.34   02    1250.
## 3  2020 S.T4   03    3511.
## 4  2020 B.35   04    1200.
## 5  2020 F.J4   05    1542.
```

Para mais, também podemos indiretamente definir a linha pela qual a função deve iniciar a leitura, por meio do argumento `skip`. Nesse argumento, você vai determinar quantas linhas do início do arquivo devem ser desconsideradas pela função. Portanto, no exemplo abaixo, eu estou ignorando as 2 primeiras linhas do arquivo `t`.

```
read_delim(t, delim = ";", skip = 2, col_names = col)
```

```
## # A tibble: 7 x 4
##   Ano Código Dia  Valor
##   <dbl> <chr> <chr> <dbl>
## 1  2020 S.T4   03    3511.
## 2  2020 B.35   04    1200.
## 3  2020 F.J4   05    1542.
## 4  2020 A.12   06    9854.
## 5  2020 B.Q2   07    7654.
## 6  2020 G.T4   08    4328.
## 7  2020 E.7A   09    2310.
```

3.7 Um estudo de caso: lendo os microdados da PNAD Contínua com `read_fwf()`

A PNAD Contínua é uma pesquisa amostral, e vem sendo realizada desde janeiro de 2012 pelo Instituto Brasileiro de Geografia e Estatística (IBGE, 2019). Os principais indicadores periódicos do mercado de trabalho são extraídos dessa pesquisa, e por isso, ela representa uma das principais fontes de informação econômica e demográfica do país. Nessa seção, vamos utilizar as funções do pacote `readr` para importarmos os microdados da divulgação trimestral dessa pesquisa para o R.

A PNAD Contínua, é organizada em três pesquisas que possuem periodicidades diferentes, são elas: PNAD Contínua Anual, PNAD Contínua Mensal e PNAD Contínua Trimestral. Em outras palavras, ao longo do ano, existem três pesquisas da PNAD Contínua, sendo construídas ao mesmo tempo. Porém, essas três pesquisas são divulgadas em períodos diferentes do ano, empregam níveis de agregação diferentes, e buscam medir variáveis demográficas diferentes. Nessa seção, vamos focar

nos microdados da divulgação trimestral da PNAD Contínua, sendo essa a principal parte da PNAD Contínua, e a mais utilizada. Você pode encontrar os microdados da PNAD Contínua Trimestral, na [página oficial da pesquisa](#), ou no endereço da [página do servidor](#), onde esses microdados estão hospedados.

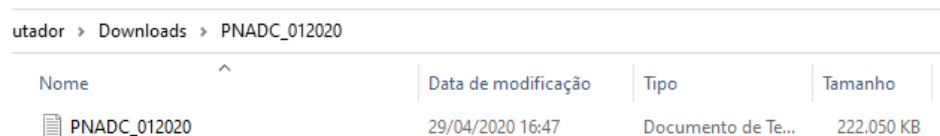
Para que você possa acompanhar os comandos mostrados nessa seção, lembre-se de chamar pelo pacote `readr`, ou pelo `tidyverse` (que contém o pacote `readr`). Como vamos utilizar o operador *pipe* (`%>%`) ao longo desse capítulo, é possível que você também tenha que chamar pelo pacote `magrittr`.

```
library(readr)
library(magrittr)
library(tidyverse)
```

3.7.1 Conhecendo a estrutura dos microdados

Antes de importarmos esses dados, precisamos conhecer a estrutura do arquivo que contém esses dados. Ou seja, precisamos saber qual a extensão desse arquivo, e de que maneira os dados estão organizados dentro desse arquivo. Como exemplo, eu fui até a [página oficial da pesquisa](#), e baixei os microdados do primeiro trimestre de 2020. O arquivo veio compactado (`.zip`), e por isso, eu o descompactei para que tivéssemos acesso ao arquivo bruto que contém os microdados, mostrado na figura 3.7.

Figura 3.7: Arquivo contendo os microdados da PNAD Contínua - 1º Trimestre de 2020



utador > Downloads > PNADC_012020			
Nome	Data de modificação	Tipo	Tamanho
 PNADC_012020	29/04/2020 16:47	Documento de Te...	222.050 KB

Fonte: Elaboração própria do autor.

Como podemos ver pela figura 3.7, o arquivo é um simples documento de texto (extensão `.txt`), e todas as funções de importação do pacote `readr` são capazes de ler este tipo de arquivo. Porém, ainda temos que identificar o tipo, ou a estrutura desse documento de texto. Em outras palavras, precisamos compreender como esses dados estão organizados dentro desse arquivo. Será que os valores de cada coluna são separados por vírgulas (`.csv`)? por ponto e vírgula (`.csv`)? por tabulação (`.tsv`)? Para descobriremos, precisamos dar uma olhada no arquivo.

Porém, o tamanho do arquivo é considerável (aproximadamente 222 MB). Isso nos dá a entender que a base de dados contida nesse arquivo, é relativamente grande. Como nós queremos dar apenas uma olhada, talvez seja mais interessante lermos apenas as 5 primeiras linhas do arquivo. As funções de

importação do pacote `readr`, geralmente possuem um argumento `n_max`, onde podemos configurar o número máximo de linhas a serem lidas do arquivo. Portanto, ao aplicarmos a função `read_csv()` abaixo, podemos ver as cinco primeiras linhas do arquivo. A primeira coisa que podemos abstrair do resultado, é que o arquivo de texto parece uma muralha de números, e aparentemente não se encaixa em nenhuma das hipóteses anteriores.

```
read_csv(
  "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
  n_max = 5
)

## # A tibble: 5 x 1
##   `202011111 11000001611100110107511000098.75663631000139.7342223000053491~`
##   <chr>
## 1 202011111 11000001611100110107511000098.75663631000139.7342223000053491~
## 2 202011111 11000001611100110107511000098.75663631000139.7342223000053491~
## 3 202011111 11000001611100110107511000098.75663631000139.7342223000053491~
## 4 202011111 11000001611100110307511000098.75663631000139.7342223000053491~
## 5 202011111 11000001611100110307511000098.75663631000139.7342223000053491~
```

Esse é um exemplo de arquivo chamado de *fixed width file* (.fwf), ou “arquivo de largura fixa”. Provavelmente, o principal motivo pelo qual o IBGE decidiu adotar esse formato de arquivo na divulgação de seus dados, está no fato de que arquivos desse tipo, são muito mais rápidos de se ler em programas, do que um arquivo CSV tradicional. Pois os valores de cada coluna em um arquivo *fixed width file*, se encontram sempre nos mesmos lugares ao longo do arquivo. Em contrapartida, esse tipo de arquivo, torna a sua vida mais difícil, pois você precisa especificar a largura, ou o número de caracteres presentes em cada coluna, para a função que será responsável por ler esse arquivo.

Ou seja, nesse tipo arquivo, não há qualquer tipo de valor ou especificação responsável por delimitar as colunas da base de dados. O arquivo simplesmente contém todos os valores, um do lado do outro. Será sua tarefa, dizer ao programa (no nosso caso, o R) quantos caracteres estão presentes em cada coluna, ou em outras palavras, definir em quais caracteres estão as “quebras” de colunas.

Isso significa, que você irá precisar de um dicionário desses dados, contendo as especificações de cada coluna dessa base de dados. No caso da PNAD Contínua, são oferecidos: 1) o dicionário das variáveis (geralmente em uma planilha do Excel, com extensão .xls), que contém uma descrição completa de cada variável (ou coluna) presente na base; 2) e o arquivo de texto input, que contém as especificações para a importação da base. Você pode baixar esses arquivos separadamente, na [página do servidor](#) em que os microdados são hospedados, ou então, você pode baixar um ZIP (Dicionario_input.zip) desses arquivos neste [link](#). Logo abaixo, na figura 3.8, temos uma foto desses arquivos em meu computador.

Entretanto, para surpresa de muitos, o arquivo de texto input (que geralmente assume o nome de `input_PNADC_trimestral.txt`), é na verdade, um *script* de importação utilizado pelo programa

Figura 3.8: Arquivos input e dicionário da PNAD Contínua

Organizar		Novo		Abrir		Selecionar	
utador > Downloads > PNADC_012020 > input							
Nome		Data de modificação		Tipo		Tamanho	
	dicionario_PNADC_microdados_trimestral	30/10/2019 11:09		Planilha do Micro...		158 KB	
	input_PNADC_trimestral.sas	08/05/2019 13:56		Arquivo SAS		15 KB	
	input_PNADC_trimestral	08/05/2019 13:56		Documento de Te...		15 KB	

Fonte: Elaboração própria do autor.

estatístico SAS¹. O SAS é um programa estatístico pago, parecido com o seu concorrente SPSS², sendo um programa mais popular no mercado americano. Logo, se você estivesse trabalhando com o programa SAS, você já teria um *script* pronto para importar os microdados da PNAD Contínua. Como não é o nosso caso, temos que extrair, a partir desse *script* de SAS, as especificações de cada coluna.

3.7.2 Extraíndo especificações de um *script* SAS

Como veremos mais a frente, extrair as especificações desse *script* é uma tarefa simples, e existem hoje, diversas ferramentas que podemos utilizar para rapidamente extraírmos essas informações do *script*, sem a necessidade de um trabalho manual. Porém, antes de partirmos para a prática, precisamos primeiro, compreender a estrutura do *script* de SAS, presente nesse arquivo *input* (input_PNADC_trimestral). Na figura 3.9, temos um resumo que descreve essa estrutura.

O *script*, ou mais especificamente, os comandos que definem a importação dos dados, se inicia pelo termo *input*, logo, estamos interessados em todas as configurações feitas após esse termo. As especificações de cada coluna, são compostas por 3 itens principais: 1) a posição inicial dessa coluna (ou a posição do caractere que inicia essa coluna); 2) o nome dessa coluna; e 3) a largura dessa coluna, ou em outras palavras, a quantidade de caracteres presentes em cada linha dessa coluna. Para o nosso objetivo, precisamos extrair os dois últimos componentes (o nome e a largura da coluna), além de definirmos se essa coluna é numérica ou textual, que é determinado pela presença ou não de um cifrão (\$) ao lado da largura da coluna, no *script*.

A melhor forma de organizarmos essas especificações, é criarmos uma tabela, onde cada linha corresponde a uma coluna dos microdados, e cada coluna dessa tabela contém uma das especificações (nome da coluna, largura da coluna, é numérica ou textual?) de cada coluna dos microdados. Para construir essa tabela, eu costumo utilizar macros de um programa de edição de texto (como o No-

¹<https://www.sas.com/en_us/home.html>

²<<https://www.ibm.com/products/spss-statistics>>

Figura 3.9: Resumo da estrutura de um script de importação do SAS

Comando inicial de importação	→	input
Caractere que inicia a coluna	→	@0001 Ano \$4.
Nome da coluna	→	@0005 Trimestre \$1.
	→	@0006 UF \$2.
	→	@0008 Capital \$2.
	→	@0010 RM_RIDE \$2.
Largura da coluna (número de caracteres presentes na coluna)	→	@0012 UPA → \$9.
	→	@0021 Estrato \$7.
	→	@0028 V1008 \$2.
Todas as colunas que contém o cifrão, devem ser interpretadas como colunas de texto	→	@0030 V1014 → \$2.
	→	@0032 V1016 \$1.
	→	@0033 V1022 \$1.
	→	@0034 V1023 \$1.
Todas as colunas que não contém o cifrão, devem ser importadas como colunas numéricas	→	@0035 V1027 → 15.
	→	@0050 V1028 15.
	→	@0065 V1029 9.
	→	.
	→	.
	→	.
	→	.

Fonte: Elaboração própria do autor.

tepad++³) sobre o arquivo `input_PNADC_trimestral.txt`, de forma a eliminar os textos irrelevantes, e arrumar as especificações na estrutura de um arquivo CSV (`.csv`). Dessa forma, eu posso importar esse arquivo CSV resultante para o R, e adquirir a tabela que desejo. Como um guia, você pode ter acesso a esse arquivo CSV, através da cópia que deixei no [Apêndice A](#).

Portanto, após extrair as especificações de cada coluna do arquivo `input`, eu tenho como resultado, um arquivo CSV chamado `widths.txt`, que eu posso ler através da função `read_csv()`. Veja pelo resultado abaixo, que eu defini três colunas nesse arquivo CSV. A coluna `variavel` possui os nomes das colunas, na ordem em que elas aparecem no *script* do arquivo `input`, e portanto, nos microdados. A coluna `width` possui o número de caracteres presentes em cada uma dessas colunas. Já a coluna `char`, possui um valor lógico, indicando se os dados contidos nessa coluna, devem ser interpretados como texto (`TRUE`), ou como números (`FALSE`).

```
col_width <- read_csv(
  "C:/Users/Pedro/Downloads/PNADC_012020/widths.txt",
  col_names = c("variavel", "width", "char")
)

## -- Column specification -----
## cols(
##   variavel = col_character(),
##   width = col_double(),
##   char = col_logical()
## )

col_width

## # A tibble: 217 x 3
##   variavel width char
##   <chr>     <dbl> <lgl>
## 1 Ano           4 TRUE
## 2 Trimestre     1 TRUE
## 3 UF            2 TRUE
## 4 Capital       2 TRUE
## 5 RM_RIDE       2 TRUE
## 6 UPA           9 TRUE
## 7 Estrato       7 TRUE
## 8 V1008         2 TRUE
## 9 V1014         2 TRUE
## 10 V1016        1 TRUE
## # ... with 207 more rows
```

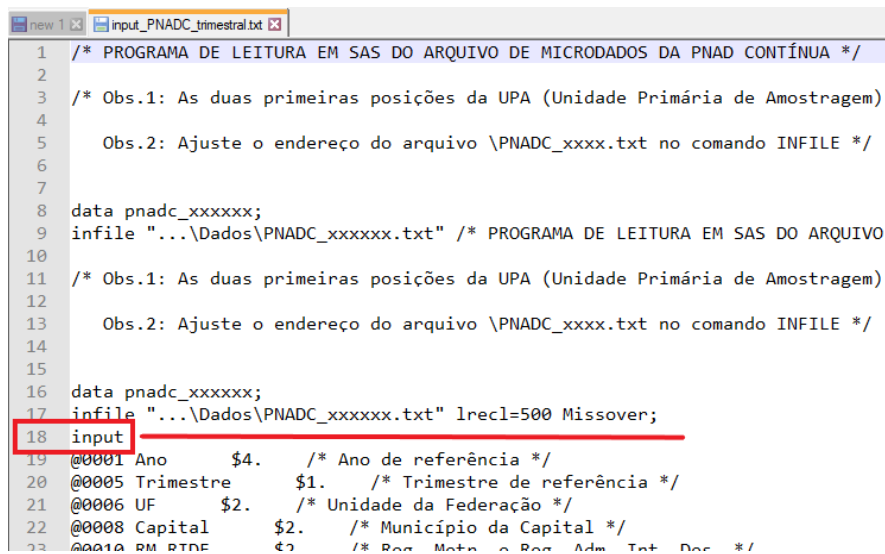
³<https://notepad-plus-plus.org/>

3.7.3 O pacote SAScii como um atalho útil

O pacote SAScii nos oferece um conjunto de funções voltadas para a importação de arquivos *fixed width file*. Porém, dentre as suas funcionalidades, o pacote também nos oferece uma função capaz de converter *scripts* de importação do programa SAS, e extrair as especificações de cada coluna em um *data.frame*. Ou seja, podemos utilizar a função `parse.SAScii()` para extraírmos as especificações de cada coluna do *script* presente no arquivo *input*.

Essa função é bem simples, e possui dois argumentos principais: 1) `sas_ri`, o endereço até o arquivo contendo o *script* de SAS a ser convertido; 2) `beginline`, a linha do arquivo em que o *script* de importação se inicia, ou em outras palavras, a linha do *script* onde o termo *input* aparece. Como podemos ver pela figura 3.10, eu abri o arquivo *input* em meu Notepad++, que possui na lateral esquerda, a numeração de cada linha. Dessa forma, eu posso rapidamente identificar que o termo *input* aparece na linha 18 do arquivo.

Figura 3.10: Início do script de importação



```

1  /* PROGRAMA DE LEITURA EM SAS DO ARQUIVO DE MICRODADOS DA PNAD CONTÍNUA */
2
3  /* Obs.1: As duas primeiras posições da UPA (Unidade Primária de Amostragem)
4
5     Obs.2: Ajuste o endereço do arquivo \PNADC_xxxx.txt no comando INFILE */
6
7
8  data pnadc_xxxxxx;
9  infile "...\\Dados\\PNADC_xxxxxx.txt" /* PROGRAMA DE LEITURA EM SAS DO ARQUIVO
10
11  /* Obs.1: As duas primeiras posições da UPA (Unidade Primária de Amostragem)
12
13     Obs.2: Ajuste o endereço do arquivo \PNADC_xxxx.txt no comando INFILE */
14
15
16  data pnadc_xxxxxx;
17  infile "...\\Dados\\PNADC_xxxxxx.txt" lrecl=500 Missover;
18  input
19  @0001 Ano      $4.      /* Ano de referência */
20  @0005 Trimestre $1.      /* Trimestre de referência */
21  @0006 UF       $2.      /* Unidade da Federação */
22  @0008 Capital  $2.      /* Município da Capital */
23  @0010 RM RTDE  $2.      /* Reg. Metr. e Reg. Adm. Int. Doc. */

```

Fonte: Elaboração própria do autor.

Com essas informações em mente, eu poderia gerar a tabela `col_width`, através dos seguintes comandos:

```

library(SAScii)
library(tibble)

col_width <- parse.SAScii(
  "C:/Users/Pedro/Downloads/PNADC_012020/input_PNADC_trimestral.txt",
  beginline = 18
)
```

```
as_tibble(col_width)
```

```
## # A tibble: 217 x 4
##   varname   width char divisor
##   <chr>     <dbl> <lgl>   <dbl>
## 1 ANO             4 TRUE     1
## 2 TRIMESTRE       1 TRUE     1
## 3 UF              2 TRUE     1
## 4 CAPITAL         2 TRUE     1
## 5 RM_RIDE         2 TRUE     1
## 6 UPA             9 TRUE     1
## 7 ESTRATO         7 TRUE     1
## 8 V1008           2 TRUE     1
## 9 V1014           2 TRUE     1
## 10 V1016          1 TRUE     1
## # ... with 207 more rows
```

3.7.4 Importando os microdados da PNAD Contínua

Agora que possuímos as especificações necessárias de cada coluna, podemos começar o processo de importação dos microdados da PNAD Contínua. Como esses microdados estão estruturados em um arquivo de texto do tipo *fixed width file* (.fwf), podemos utilizar a função `read_fwf()` para ler o arquivo. Pois como o próprio nome dessa função dá a entender, ela é especializada nesse tipo de arquivo.

O primeiro argumento (file) dessa função, é o caminho até o arquivo a ser importado. Já o segundo argumento (col_positions), será o local onde vamos fornecer as especificações de cada coluna. Entretanto, nós precisamos utilizar uma função como a `fwf_widths()`, para definirmos essas especificações no argumento `col_positions`. Na função `fwf_widths()` temos apenas dois argumentos, que são `widths` e `col_names`. Basta fornecermos ao argumento `widths`, as larguras de cada coluna, e ao argumento `col_names`, os nomes de cada coluna, como no exemplo abaixo.

```
pnad_continua <- read_fwf(
  "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
  col_positions = fwf_widths(col_width$width, col_names = col_width$variavel)
)

## -- Column specification -----
## cols(
##   .default = col_double(),
##   RM_RIDE = col_logical(),
##   V1008 = col_character(),
##   V1014 = col_character(),
##   V1027 = col_character(),
##   V1028 = col_character(),
```

```
## V1029 = col_character(),
## V2001 = col_character(),
## V2003 = col_character(),
## V2005 = col_character(),
## V2008 = col_character(),
## V20081 = col_character(),
## V2009 = col_character(),
## `3003` = col_logical(),
## V3003A = col_character(),
## V3004 = col_logical(),
## V3005 = col_logical(),
## V3006 = col_character(),
## V3009 = col_logical(),
## V3009A = col_character(),
## V3011 = col_logical()
## # ... with 87 more columns
## )
## i Use `spec()` for the full column specifications.
## Warning: 156486 parsing failures.
## row col expected actual file
## 1670 V40431 1/0/T/F/TRUE/FALSE 2 'C:/Users/Pedro/Downloads/PNADC~'
## 2194 V4057 1/0/T/F/TRUE/FALSE 2 'C:/Users/Pedro/Downloads/PNADC_~'
## 2194 V405811 1/0/T/F/TRUE/FALSE 3 'C:/Users/Pedro/Downloads/PNADC~'
## 2194 V405812 1/0/T/F/TRUE/FALSE 00001200 'C:/Users/Pedro/Downloads/PNADC~'
## 2194 V405912 1/0/T/F/TRUE/FALSE 00000000 'C:/Users/Pedro/Downloads/PNADC~'
## ....
## See problems(...) for more details.
```

Como podemos ver acima, pela mensagem de *parsing failures*, obtivemos alguns problemas durante a importação. Isso ocorre, pois a função `read_fwf()` está tendo que adivinhar sozinha, quais são os tipos de dados contidos em cada coluna dos microdados. Lembre-se que por padrão, se não fornecermos uma descrição dos tipos de dados de cada coluna à qualquer função do pacote `readr`, essas funções vão automaticamente ler as 1000 primeiras linhas de cada coluna, e se basear nesses 1000 valores para determinar o tipo de dado incluso em cada coluna do arquivo.

Esse sistema automático, apesar de útil, se torna frágil a medida em que o tamanho da nossa base cresce. Pois essas 1000 linhas vão representar uma parte cada vez menor da base, e portanto, podem não ser suficientes para determinar com precisão o tipo de dado contido em cada coluna. No nosso exemplo, a base da PNAD possui 487 mil linhas, logo, essas 1000 linhas representam apenas 0,2% da base. Se a função não está sendo capaz de adivinhar corretamente, os tipos de dados de cada coluna, nós precisamos dizer a ela exatamente quais são esses tipos. Para isso, vamos utilizar os dados contidos na coluna `char`, da nossa tabela `col_width`.

As funções de importação do pacote `readr`, possuem o argumento `col_types`, onde podemos definir os tipos de cada coluna. Essa definição pode ser fornecida, utilizando-se a função `cols()`. Porém, para o nosso caso, creio que será mais prático, utilizarmos um método alternativo que o

Primeiro, precisamos construir esse vetor de letras, que indicam o tipo de cada coluna. Com os dados da nossa tabela `col_width`, nós já sabemos que todo valor `TRUE` na coluna `char`, indica uma coluna de texto, e portanto, essa coluna deve ser interpretada como uma coluna do tipo `character`. Já os valores `FALSE` indicam uma coluna numérica, e por isso, essa coluna deve ser interpretada como uma coluna do tipo `double`. Com isso, podemos utilizar a função `ifelse()`, para construirmos um vetor inicial de letras, baseado nos valores da coluna `char`. Em seguida, podemos juntar todas essas letras em um *string* só, com a função `paste()`.

Agora com o vetor tipos, podemos fornecê-lo ao argumento `col_types` e realizar novamente o processo de importação, com os tipos das colunas sendo corretamente interpretados. Porém, repare que mesmo definindo os tipos das colunas, obtivemos novamente erros durante o processo de importação. Dessa vez, foram mais de 2 milhões de erros. Isso não significa necessariamente que o nosso processo de importação esteja incorretamente especificado. Porém, nós deveríamos pelo menos compreender o porque esses erros ocorrem.

```
pnad_continua <- read_fwf(
  "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
  col_positions = fwf_widths(col_width$width, col_names = col_width$variavel),
  col_types = tipos
)

## Warning: 2032039 parsing failures.
## row    col expected actual          file
##   1 VD4032 a double      . 'C:/Users/Pedro/Downloads/PNADC_012020/PNADC_01~'
##   1 VD4033 a double      . 'C:/Users/Pedro/Downloads/PNADC_012020/PNADC_01~'
##   1 VD4034 a double      . 'C:/Users/Pedro/Downloads/PNADC_012020/PNADC_01~'
##   2 VD4031 a double      . 'C:/Users/Pedro/Downloads/PNADC_012020/PNADC_01~'
##   2 VD4032 a double      . 'C:/Users/Pedro/Downloads/PNADC_012020/PNADC_01~'
## ... ..
## See problems(...) for more details.
```


3.7.5 Analisando erros de importação

Nós podemos obter através da função `problems()`, uma tabela contendo todos os erros que ocorreram durante esse processo de importação. Precisamos apenas fornecer a essa função, os comandos que geraram esses problemas, como no exemplo abaixo.

```
problemas <- problems(
  read_fwf(
    "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
    col_positions = fwf_widths(col_width$width, col_names = col_width$variavel),
    col_types = tipos
  )
)

problemas

## # A tibble: 2,032,039 x 5
##   row col      expected actual file
##   <int> <chr>    <chr>      <chr> <chr>
## 1     1 VD4032 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 2     1 VD4033 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 3     1 VD4034 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 4     2 VD4031 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 5     2 VD4032 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 6     2 VD4033 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 7     2 VD4034 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 8     2 VD4035 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 9     3 VD4031 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## 10    3 VD4032 a double    . 'C:/Users/Pedro/Downloads/PNADC_012020/PNAD~
## # ... with 2,032,029 more rows
```

Pelo que podemos ver da coluna `actual`, parece que os erros estão ocorrendo, pela presença de um ponto final (“.”), nos locais em que deveriam estar números (`double`). Podemos utilizar a função `unique()` sobre a coluna `actual` para identificarmos se há algum outro problema que precisamos analisar. Pelo resultado abaixo, percebemos que todos os mais de 2 milhões de erros gerados, estão sendo causados por essa presença de pontos finais na base. Também podemos utilizar a função `unique()` sobre a coluna `col`, para descobrirmos em quais colunas esse erro ocorre. Vemos abaixo, que esses erros estão concentrados em cinco das últimas colunas de toda a base (a última coluna da base é VD4037).

```
unique(problemas$actual)

## [1] "."

unique(problemas$col)

## [1] "VD4032" "VD4033" "VD4034" "VD4031" "VD4035"
```

Seria uma boa ideia, olharmos mais de perto como essas colunas aparecem no arquivo de microdados. Para determinarmos a parte do arquivo que diz respeito a essas colunas, precisamos descobrir o intervalo de caracteres que cobrem essas colunas, através dos dados da tabela `col_width`. Para isso, vamos precisar descobrir o número total de caracteres em cada linha (ou em outras palavras, a largura total da base), ao somarmos a largura de todas as colunas na tabela `col_width`. Ao longo do caminho, teremos que subtrair uma faixa desse total, para descobrirmos o caractere que inicia o intervalo de colunas que estamos interessados.

```
(total_caracteres <- sum(col_width$width))
```

```
## [1] 464
```

Em seguida, podemos aplicar a função `tail()` sobre a tabela `col_width`, para extraírmos as últimas linhas dessa tabela, e verificarmos as especificações das colunas que cobrem essa faixa. Pois nós sabemos que as variáveis que geraram problemas na importação, estão entre as últimas colunas dos microdados, logo, as especificações dessas colunas vão se encontrar nas últimas linhas da tabela `col_width`. Vemos abaixo, que as duas últimas colunas da base (VD4036 e VD4037), das quais não estamos interessados, possuem juntas, 2 caracteres de largura. Portanto, o intervalo que cobre as colunas que geraram os problemas na importação (VD4031-VD4035), termina no 462º caractere, como vemos abaixo. Pelo resultado de `tail()`, vemos que as colunas das quais estamos interessados (VD4031-VD4035), somam 15 caracteres de largura. Tendo isso em mente, o intervalo que cobre essas colunas, se inicia no 448º caractere.

```
tail(col_width, 7)
```

```
##      varname width char divisor
## 211  VD4031     3 FALSE        1
## 212  VD4032     3 FALSE        1
## 213  VD4033     3 FALSE        1
## 214  VD4034     3 FALSE        1
## 215  VD4035     3 FALSE        1
## 216  VD4036     1  TRUE        1
## 217  VD4037     1  TRUE        1
```

```
(fim_intervalo <- total_caracteres - 2)
```

```
## [1] 462
```

```
(inicio_intervalo <- fim_intervalo - 15 + 1)
```

```
## [1] 448
```

Portanto, nós temos agora a posição dos caracteres que iniciam e terminam o intervalo de caracteres que dizem respeito as colunas que estamos interessados. Porém, ainda precisamos calcular os caracteres de início e de fim de cada uma das cinco colunas (VD4031-VD4035), que cobrem esse intervalo. Para esse trabalho, podemos aplicar uma simples aritmética, como a aplicada pelo código abaixo.

```
(inicio <- (0:4 * 3) + inicio_intervalo)
```

```
## [1] 448 451 454 457 460
```

```
(fim <- (1:5 * 3) + inicio_intervalo - 1)
```

```
## [1] 450 453 456 459 462
```

Agora que nós temos as posições dos caracteres que iniciam e que terminam cada uma das cinco colunas, podemos importar apenas essas cinco colunas ao R. Para isso, podemos usar novamente a função `read_fwf()`, aliada à função `fwf_positions()`. Ou seja, utilizamos anteriormente a função `fwf_widths()` para determinarmos as especificações de todas as colunas da base. Porém, como nós queremos importar apenas uma parte dessa base, vamos utilizar a função `fwf_positions()` para determinarmos as especificações dessas colunas desejadas.

Na função `fwf_positions()`, temos três argumentos principais: 1) `start`, um vetor contendo as posições dos caracteres que iniciam cada coluna; 2) `end`, um vetor contendo as posições dos caracteres que terminam cada coluna; 3) `col_names`, um vetor contendo os nomes dessas colunas selecionadas. Tendo esses argumentos em mente, podemos importar as cinco colunas da seguinte maneira:

```
colunas <- c("VD4031", "VD4032", "VD4033", "VD4034", "VD4035")
```

```
conferir <- read_fwf(
  "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
  col_positions = fwf_positions(
    start = inicio,
    end = fim,
    col_names = colunas
  )
)
```

```
## -- Column specification -----
## cols(
##   VD4031 = col_character(),
##   VD4032 = col_character(),
##   VD4033 = col_character(),
##   VD4034 = col_character(),
##   VD4035 = col_character()
## )
```

Logo abaixo, temos o resultado do intervalo que selecionamos do arquivo, em que podemos ver o grupo de pontos finais que estão causando o problema. Agora, temos que identificar o motivo desses pontos estarem aí. Se nós retornarmos às especificações dessas colunas apresentadas na tabela `col_width`, nós sabemos que essas colunas são colunas numéricas. Será que esses pontos estão aí, para marcar as casas decimais dos números dessa coluna?

Talvez não seja esse o caso. Pois se esses pontos estivessem de fato, marcando as casas decimais, porque eles não aparecem na primeira linha das colunas VD4031 e VD4035? Isto é, por que o valor 040 que aparece nessas colunas, não se apresenta como 0.40, ou 04.0, ou 40.0 na tabela conferir? Lembre-se que os valores da tabela conferir, são apresentados exatamente da forma como eles se apresentam no arquivo dos microdados, pois todas essas colunas estão sendo interpretadas como `character`. Ou seja, esses valores que eram meros textos no arquivo dos microdados, continuam sendo textos no R, de forma que não houve nenhuma conversão desses valores.

```
conferir
```

```
## # A tibble: 487,937 x 5
##   VD4031 VD4032 VD4033 VD4034 VD4035
##   <chr>  <chr>  <chr>  <chr>  <chr>
## 1 040    .      .      .      040
## 2 .      .      .      .      .
## 3 .      .      .      .      .
## 4 .      .      .      .      .
## 5 .      .      .      .      .
## 6 040    .      .      .      040
## 7 .      .      .      .      .
## 8 .      .      .      .      .
## 9 .      .      .      .      .
## 10 040   .      .      .      040
## # ... with 487,927 more rows
```

Pela visão que temos até o momento, parece que as colunas VD4032, VD4033 e VD4034, estão vazias, de forma que elas possuem apenas pontos finais em toda a sua extensão. Talvez seja o momento de verificarmos essa hipótese. Podemos fazer isso, novamente por meio da função `unique()`. Pelos resultados abaixo, as colunas VD4032, VD4033 e VD4034 estão de fato vazias. Com isso, temos a seguinte questão: por que uma coluna numérica está preenchida com pontos? Se esses pontos não estão marcando as casas decimais em cada linha, é mais provável que esses pontos estejam ali simplesmente para marcar um valor vazio, ou uma observação que não pôde ser mensurada.

```
unique(conferir$VD4032)
```

```
## [1] "."
```

```
unique(conferir$VD4033)
```

```
## [1] "."
```

```
unique(conferir$VD4034)
## [1] "."
```

Em resumo, nós sabemos pelas especificações das colunas presentes no arquivo input, que as colunas VD4032, VD4033 e VD4034 devem ser interpretadas como colunas numéricas. Ao que tudo indica, esses pontos não possuem o propósito de delimitar as casas decimais. Seria apropriado encontrarmos alguma documentação que nos pudesse guiar sobre esses questionamentos. Porém, até onde pesquisei, não há qualquer menção a esses pontos ao longo da documentação do IBGE sobre esses microdados. Com as informações que possuímos, só podemos inferir que esses valores estão servindo para marcar valores não-disponíveis (em outras palavras, estão cumprindo o papel de um valor NA) nessas colunas.

Tendo essas considerações em mente, todos esses pontos presentes nessas colunas, devido ao erro que eles incorrem durante o processo de importação, serão convertidos para valores NA ao importarmos a base, e portanto, vão representar observações não-disponíveis na base. Ou seja, se a função `read_fwf()` não consegue interpretar corretamente um valor, ele acaba sendo convertido para um valor NA.

```
pnad_continua <- read_fwf(
  "C:/Users/Pedro/Downloads/PNADC_012020/PNADC_012020.txt",
  col_positions = fwf_widths(col_width$width, col_names = col_width$variavel),
  col_types = tipos
)
```

3.8 Exportando os seus dados com o pacote readr

Mais do que importar os seus dados para dentro do R, haverá um momento em que você deseja exportar os seus resultados para fora do R, de forma que você possa enviá-los para os seus colegas de trabalho ou para utilizá-los em outros programas. Em um momento como esse, você deseja escrever um arquivo estático em seu computador, contendo esses resultados. O pacote `readr` oferece funções que permitem a escrita de um conjunto de arquivos de texto. Logo abaixo, temos uma lista relacionando os tipos de arquivos de texto às respectivas funções do pacote:

- 1) `write_csv2()`: constrói um arquivo CSV, segundo o padrão adotado por alguns países europeus; utilizando pontos e vírgulas (;) como separador.
- 2) `write_csv()`: constrói um arquivo CSV, segundo o padrão americano; utilizando vírgulas (,) como separador.
- 3) `write_tsv()`: constrói um arquivo TSV.
- 4) `write_delim()`: função geral onde você pode definir o caractere a ser utilizado como separador no arquivo de texto construído.

Um fator muito importante sobre o pacote `readr` em geral, é que todas as suas funções utilizam o *encoding* UTF-8 o tempo todo. Logo, ao utilizar essas funções para exportar os seus dados,

lembre-se sempre que os arquivos construídos por essas funções vão estar utilizando o *encoding* UTF-8. Isso significa que ao utilizar esses arquivos em outros programas como o Excel, você precisa informar ao programa para utilizar o *encoding* UTF-8 ao ler o arquivo.

Para além disso, você não terá nenhum outro problema com esses arquivos. Porém, caso você se sinta incomodado com esse comportamento, você pode utilizar as variantes dessas funções presentes nos pacotes básicos do R (`write.csv2()`, `write.csv()`, `write.table()`). Pois essas funções variantes vão escrever o arquivo definido, de acordo com o *encoding* padrão de seu sistema.

O primeiro argumento (`x`) dessas funções, se trata do nome do objeto em sua sessão que contém os dados que você deseja exportar. Já no segundo argumento (`file`) dessas funções, você deve definir o nome do novo arquivo estático que será construído. Por exemplo, se eu possuo uma tabela chamada, e desejo salvá-la em um arquivo chamado `transf.csv`, eu preciso construir os seguintes comandos:

```
write_csv2(transf, file = "transf.csv")
```

Após executar os comandos acima, você irá encontrar na pasta que representa o seu diretório de trabalho atual no R, um novo arquivo chamado `transf.csv` que contém os seus dados. Vale destacar, que você pode salvar esse novo arquivo em diferentes áreas de seu computador. Basta que você forneça um endereço (absoluto ou relativo) até a pasta desejada, em conjunto com o nome do novo arquivo. Como exemplo, eu posso salvar a tabela `Censo_2010` dentro da minha área de trabalho da seguinte forma:

```
write_csv2(Censo_2010, file = "C:/Users/Pedro/Desktop/Censo_2010.csv")
```

3.9 Importando planilhas do Excel com readxl

O Excel continua sendo um dos programas mais populares no mundo e, por essa razão, muitas pessoas ainda o utilizam para analisar dados e gerar gráficos. Tendo isso em vista, nessa seção, vamos aprender como podemos importar para o R, dados que se encontram em planilhas do Excel (`.xlsx`), através da função `read_excel()` que pertence ao pacote `readxl`.

O principal argumento da função `read_excel()` corresponde novamente ao endereço até o arquivo que você deseja ler, ou apenas o seu nome caso esse arquivo se encontre em seu diretório de trabalho atual.

```
library(readxl)
```

```
codigos <- read_excel("codigos.xlsx")
```

```
codigos
```

```
## # A tibble: 853 x 4
##   IBGE1 IBGE2 SEF Municipios
##   <dbl> <dbl> <dbl> <chr>
## 1 310010    10    1 ABADIA DOS DOURADOS
## 2 310020    20    2 ABAETÉ
## 3 310030    30    3 ABRE CAMPO
## 4 310040    40    4 ACAIACA
## 5 310050    50    5 AÇUCENA
## 6 310060    60    6 ÁGUA BOA
## 7 310070    70    7 ÁGUA COMPRIDA
## 8 310080    80    8 AGUANIL
## 9 310090    90    9 ÁGUAS FORMOSAS
## 10 310100   100   10 ÁGUAS VERMELHAS
## # ... with 843 more rows
```

3.9.1 Delimitando a parte de seu arquivo .xlsx

Um único arquivo .xlsx pode conter várias planilhas, ou várias abas (*sheet's*) diferentes. Por padrão, a função `read_excel()` sempre lê a primeira planilha de seu arquivo .xlsx. Porém, você pode ler diferentes planilhas de seu arquivo por meio do argumento `sheet`. Somos capazes de selecionar a planilha desejada de acordo com a sua ordem (1, 2, 3, ...), ou de acordo com o nome dado à aba que a contém.

```
## Lê a terceira planilha do arquivo
read_excel("datasets.xlsx", sheet = 3)
```

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

```
## Lê a planilha presente na aba denominada mtcars
read_excel("datasets.xlsx", sheet = "mtcars")
```

```
## # A tibble: 32 x 11
##   mpg   cyl  disp    hp  drat    wt  qsec    vs  am  gear  carb
```

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

Além dessas configurações, conseguimos delimitar o intervalo de células a serem lidas pela função, através do argumento `range`. Podemos fornecer esse intervalo em dois estilos diferentes. Nós podemos utilizar o sistema tradicional do Excel (CL:CL), como no exemplo abaixo, em que estamos lendo da célula A1 à célula C150 através da notação A1:C150.

```
read_excel("datasets.xlsx", range = "A1:C150")
```

```
## # A tibble: 149 x 3
##   Sepal.Length Sepal.Width Petal.Length
##         <dbl>         <dbl>         <dbl>
## 1         5.1         3.5         1.4
## 2         4.9         3         1.4
## 3         4.7         3.2         1.3
## 4         4.6         3.1         1.5
## 5         5         3.6         1.4
## 6         5.4         3.9         1.7
## 7         4.6         3.4         1.4
## 8         5         3.4         1.5
## 9         4.4         2.9         1.4
## 10        4.9         3.1         1.5
## # ... with 139 more rows
```

Uma outra possibilidade é utilizarmos as funções `cell_cols()` e `cell_rows()` que limitam o intervalo para apenas uma das dimensões da planilha. Ou seja, nós empregamos a função `cell_cols()`, quando desejamos ler todas as linhas, e, apenas algumas colunas da planilha. Enquanto com a função `cell_rows()`, desejamos ler todas as colunas da tabela, porém, queremos extrair apenas uma parte das linhas.

As colunas de uma planilha do Excel, são identificadas por uma letra ou por um conjunto de letras (ex: A; E; F; BC). Por isso, ao utilizar a função `cell_cols()` você pode delimitar as colunas a serem lidas de duas formas: 1) utilizando a notação do Excel (C:C), com as letras que representam as colunas desejadas; 2) ou através de um vetor numérico que representa a ordem das colunas, e contém o intervalo desejado.


```
## Da coluna A até a coluna C
read_excel("datasets.xlsx", range = cell_cols("A:C"))
```

```
## Da 1° até a 3° coluna
read_excel("datasets.xlsx", range = cell_cols(1:3))
```

Por outro lado, para delimitarmos o intervalo de linhas em `cell_rows()` precisamos apenas fornecer um vetor de dois elementos, contendo os limites superior e inferior do intervalo, ou então, uma sequência que cobre esses limites.

```
## Da 1° até a 140° linha
read_excel("datasets.xlsx", range = cell_rows(1:140))
```

```
## Da 10° até a 400° linha
read_excel("datasets.xlsx", range = cell_rows(c(10, 400)))
```

O argumento `range` é tão flexível que nós podemos utilizá-lo para executar o trabalho do argumento `sheet`. Isto é, além do intervalo de células, nós também podemos selecionar a aba do arquivo `.xlsx` a ser lida pela função, através do argumento `range`. No Excel, quando você está utilizando em sua planilha, algum valor que é proveniente de uma outra planilha do mesmo arquivo `.xlsx`, o Excel cria uma referência até esse valor. Essa referência possui o nome da planilha em conjunto com a referência da célula onde o valor se encontra, separados por um ponto de exclamação (!). Logo, se eu quisesse ler da célula A1 até a célula C150, da planilha denominada `mtcars`, do arquivo `datasets.xlsx`, eu precisaria criar a seguinte referência no argumento `range`:

```
read_excel("datasets.xlsx", range = "mtcars!A1:C150")
```

```
## # A tibble: 149 x 3
##   mpg   cyl  disp
##   <dbl> <dbl> <dbl>
## 1  21     6  160
## 2  21     6  160
## 3 22.8     4  108
## 4 21.4     6  258
## 5 18.7     8  360
## 6 18.1     6  225
## 7 14.3     8  360
## 8 24.4     4  147.
## 9 22.8     4  141.
## 10 19.2     6  168.
## # ... with 139 more rows
```

Apesar de sua flexibilidade, o argumento `range` pressupõe que você conheça exatamente as células que compõe os limites de sua tabela, ou então, que você pelo menos tenha uma boa compreensão de

onde eles se encontram. Por isso, você também possui na função `read_excel()` os argumentos `skip` e `n_max`, que funcionam exatamente da mesma forma empregada pelas funções do pacote `readr`. Logo, esses argumentos representam uma alternativa menos flexível, mas, talvez sejam mais ideais para as suas necessidades, especialmente se você deseja apenas pular algumas linhas de metadados que se encontram no início de sua planilha.

```
read_excel("datasets.xlsx", sheet = 2, n_max = 50, skip = 10, col_names = FALSE)
```

```
## # A tibble: 23 x 11
##   ...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...11
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  19.2     6 168.   123  3.92  3.44  18.3     1     0     4     4
## 2  17.8     6 168.   123  3.92  3.44  18.9     1     0     4     4
## 3  16.4     8 276.   180  3.07  4.07  17.4     0     0     3     3
## 4  17.3     8 276.   180  3.07  3.73  17.6     0     0     3     3
## 5  15.2     8 276.   180  3.07  3.78  18      0     0     3     3
## 6  10.4     8 472    205  2.93  5.25  18.0     0     0     3     4
## 7  10.4     8 460    215  3     5.42  17.8     0     0     3     4
## 8  14.7     8 440    230  3.23  5.34  17.4     0     0     3     4
## 9  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
## 10 30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
## # ... with 13 more rows
```

3.9.2 Definindo os tipos de dados contidos em cada coluna

Por padrão, a função `read_excel()` vai automaticamente decifrar os tipos de dados contidos em cada coluna. Porém, diferentemente das funções do pacote `readr`, que constroem essa suposição com base nos dados em si do arquivo, a função `read_excel()` adivinha os dados contidos em cada coluna, com base nos tipos associados a cada célula da planilha. Ou seja, se as células de uma coluna estão associadas ao tipo Texto, essa coluna será transformada no R em uma coluna do tipo `character`.

Pelo fato do Excel tratar cada célula de forma individual, você possui uma liberdade muito grande no programa. Por exemplo, você pode misturar dados de diferentes tipos em uma mesma coluna, ou em uma mesma linha de uma planilha do Excel. Porém, essa liberdade tem o seu preço. Um programa que trata as suas células dessa maneira, gera uma estrutura inconsistente em seus dados. Esse fato é importante, pois você tem um trabalho muito maior ao replicar cálculos em sua tabela. Com uma estrutura inconsistente, você precisa pensar não apenas em quais tipos estão associados a cada coluna de sua tabela, mas também, em quais tipos estão associados a cada célula de cada coluna. As chances de erros serem gerados durante o processo, são bem maiores.

Portanto, o sistema que a função `read_excel()` adota, está de acordo com essa característica. Pois se diversas células em uma mesma coluna possuírem tipos diferentes associados a elas, a função será capaz de reconhecer essa inconsistência, e agir adequadamente. Nós sabemos que o R leva muito a sério a consistência de seus dados, especialmente se tratando de vetores com suas regras

de coerção e, por isso, tal liberdade presente em programas como o Excel, representam um desafio para a importação de dados provenientes dessa plataforma.

No R, há duas maneiras principais de lidarmos com essa possível inconsistência de uma planilha do Excel. Uma está no uso do tipo `character`, pois esse é o tipo de dado mais flexível de todos e, portanto, consegue guardar qualquer outro tipo de dado. Outra está na adoção de listas para qualquer coluna que apresente essa inconstância.

Portanto, em toda coluna que possui dados de diferentes tipos em suas células, a função `read_excel()` vai geralmente transformar essa coluna, em uma coluna do tipo `character`. Veja no exemplo abaixo, mais especificamente, na coluna `value` que contém ao menos três tipos de dados diferentes.

```
read_excel(readxl_example("clippy.xlsx"))
```

```
## # A tibble: 4 x 2
##   name          value
##   <chr>         <chr>
## 1 Name         Clippy
## 2 Species      paperclip
## 3 Approx date of death 39083
## 4 Weight in grams    0.9
```

Como destacamos, uma outra alternativa, seria transformarmos essa coluna em uma lista. Dessa forma, nós podemos incluir qualquer tipo de dado em cada elemento dessa lista (ou em cada “célula” dessa coluna). Porém, teremos que pedir explicitamente a função `read_excel()` que realize esse tipo de transformação, através do argumento `col_types`.

Portanto, em todas as ocasiões que você precisar evitar que a função `read_excel()` decifre os tipos os tipos de cada coluna, você pode definir de forma explícita esses tipos no argumento `col_types`. Você precisa apenas fornecer um vetor a esse argumento, contendo rótulos que representam os tipos de cada coluna na ordem em que elas aparecem na planilha. Os rótulos possíveis nesse argumento são: `"skip"`, `"guess"`, `"logical"`, `"numeric"`, `"date"`, `"text"` e `"list"`.

```
read_excel(readxl_example("clippy.xlsx"), col_types = c("text", "list"))
```

```
## # A tibble: 4 x 2
##   name          value
##   <chr>         <list>
## 1 Name         <chr [1]>
## 2 Species      <chr [1]>
## 3 Approx date of death <dtm [1]>
## 4 Weight in grams    <dbl [1]>
```

3.10 Importando arquivos do SPSS, Stata e SAS com o pacote haven

Apesar de serem programas mais populares em mercados específicos, especialmente o mercado americano, algumas pessoas no Brasil ainda utilizam programas como o Stata para produzirem as suas pesquisas. Por isso, nessa seção, vamos utilizar as funções do pacote haven, com o objetivo de importarmos dados que estejam presentes em arquivos produzidos por um desses três programas: SPSS (.sav, .zsav, .por), Stata (.dta) e SAS (.sas7bdat, .sas7bcat). Logo abaixo, temos uma lista relacionando as funções do pacote com os respectivos formatos de arquivo.

- 1) `read_dta()` - Stata (.dta).
- 2) `read_spss()` - SPSS (.sav, .zsav, .por).
- 3) `read_sas()` - SAS (.sas7bdat, .sas7bcat).

Assim como as funções de importações vistas até o momento, o primeiro argumento das três funções acima, se trata do endereço ou do nome do arquivo (caso ele se encontre em seu diretório de trabalho atual) que você deseja ler.

```
read_spss("survey.sav")
```

```
read_sas("survey.sas7bdat")
```

```
read_dta("pnad_2015.dta")
```

3.10.1 Tratando variáveis rotuladas

Os programas SPSS, SAS e Stata permitem, e muitas vezes utilizam, um sistema de rótulos sobre seus valores. O uso desses rótulos é especialmente comum em colunas que representam variáveis qualitativas (cor, sexo, faixa etária, etc.). Nessas colunas, os dados são representados por valores numéricos, porém, esses valores são rotulados com um valor textual que corresponde a faixa, ou a categoria a qual aquele valor numérico corresponde.

Como exemplo, veja a tabela abaixo, ou mais especificamente, as colunas sex, marital e child. Perceba que essas três colunas, estão sendo tratadas como colunas do tipo double + labelled (dbl + lbl). Ou seja, os dados presentes nessas colunas, são dados numéricos (double). Porém, certos rótulos (labelled) estão associados a cada um desses valores. Por exemplo, todo valor igual a 1 na coluna child, indica que a pessoa entrevistada naquela linha é responsável por alguma criança (YES), enquanto todo valor igual a 2, representa uma pessoa que não possui uma criança sobre a sua tutela (NO).

```
pesquisa <- read_spss("survey.sav")
```

```
pesquisa
```

```
## # A tibble: 439 x 9
##       id      sex  age marital  child    educ    source    smoke smokenum
##   <dbl> <dbl+lb> <dbl> <dbl+lb> <dbl+1> <dbl+lb> <dbl+lb> <dbl+1>    <dbl>
## 1  415 2 [FEMA~  24 4 [MARR~ 1 [YES] 5 [COMP~ 7 [LIF~ 2 [NO]      NA
## 2    9 1 [MALE~  39 3 [LIVI~ 1 [YES] 5 [COMP~ 1 [WOR~ 1 [YES]      2
## 3  425 2 [FEMA~  48 4 [MARR~ 1 [YES] 2 [SOME~ 4 [CHI~ 2 [NO]      NA
## 4  307 1 [MALE~  41 5 [REMA~ 1 [YES] 2 [SOME~ 1 [WOR~ 2 [NO]      0
## 5  440 1 [MALE~  23 1 [SING~ 2 [NO] 5 [COMP~ 1 [WOR~ 2 [NO]      0
## 6  484 2 [FEMA~  31 4 [MARR~ 1 [YES] 5 [COMP~ 7 [LIF~ 2 [NO]      NA
## 7  341 2 [FEMA~  30 6 [SEPA~ 2 [NO] 4 [SOME~ 8 [MON~ 2 [NO]      0
## 8  300 1 [MALE~  23 2 [STEAM~ 2 [NO] 5 [COMP~ 1 [WOR~ 1 [YES]    100
## 9   61 2 [FEMA~  18 2 [STEAM~ 2 [NO] 2 [SOME~ 2 [SPO~ 1 [YES]     40
## 10  24 1 [MALE~  23 1 [SING~ 2 [NO] 6 [POST~ NA      2 [NO]      0
## # ... with 429 more rows
```

Toda coluna que estiver rotulada no arquivo, será importada dessa maneira para o R, criando um tipo misto. Porém, após a importação dos dados, o ideal é que você sempre transforme essas colunas “mistas” para o tipo factor, pois esse tipo de dado apresenta um suporte muito melhor ao longo da linguagem R. Tal transformação pode ser facilmente gerada através da função `as_factor()`, que provém do pacote `forcats`.

```
library(forcats)
```

```
pesquisa <- as_factor(pesquisa)
```

```
pesquisa
```

```
## # A tibble: 439 x 9
##       id sex      age marital  child educ      source    smoke smokenum
##   <dbl> <fct> <dbl> <fct>    <fct> <fct>    <fct>    <fct>    <dbl>
## 1  415 FEMAL~  24 MARRIED FI~ YES  COMPLETED~ LIFE IN ~ NO      NA
## 2    9 MALES  39 LIVING WIT~ YES  COMPLETED~ WORK      YES      2
## 3  425 FEMAL~  48 MARRIED FI~ YES  SOME SECO~ CHILDREN NO      NA
## 4  307 MALES  41 REMARRIED YES  SOME SECO~ WORK      NO      0
## 5  440 MALES  23 SINGLE    NO  COMPLETED~ WORK      NO      0
## 6  484 FEMAL~  31 MARRIED FI~ YES  COMPLETED~ LIFE IN ~ NO      NA
## 7  341 FEMAL~  30 SEPARATED NO  SOME ADDI~ MONEY/FI~ NO      0
## 8  300 MALES  23 STEADY REL~ NO  COMPLETED~ WORK      YES    100
## 9   61 FEMAL~  18 STEADY REL~ NO  SOME SECO~ SPOUSE O~ YES     40
## 10  24 MALES  23 SINGLE    NO  POSTGRADU~ <NA>    NO      0
## # ... with 429 more rows
```

3.10.2 Delimitando partes do arquivo

Todas as três funções do pacote `haven` possuem os argumentos `skip` e `n_max`, que novamente, funcionam da mesma forma que é empregado pelas funções do pacote `readr`. Portanto, o argumento `skip` e `n_max` definem o número de linhas a serem ignoradas no início do arquivo, e o número máximo de linhas do arquivo a serem lidas, respectivamente.

```
read_spss("survey.sav", skip = 5)
```

```
## # A tibble: 434 x 9
##   id      sex age marital  child   educ  source  smoke smokenum
##   <dbl> <dbl+lb> <dbl> <dbl+lb> <dbl+1> <dbl+lb> <dbl+lb> <dbl+1> <dbl>
## 1  484 2 [FEMA~ 31 4 [MARR~ 1 [YES] 5 [COMP~ 7 [LIF~ 2 [NO]      NA
## 2  341 2 [FEMA~ 30 6 [SEPA~ 2 [NO] 4 [SOME~ 8 [MON~ 2 [NO]      0
## 3  300 1 [MALE~ 23 2 [STEA~ 2 [NO] 5 [COMP~ 1 [WOR~ 1 [YES]    100
## 4   61 2 [FEMA~ 18 2 [STEA~ 2 [NO] 2 [SOME~ 2 [SPO~ 1 [YES]    40
## 5   24 1 [MALE~ 23 1 [SING~ 2 [NO] 6 [POST~ NA      2 [NO]      0
## 6  138 1 [MALE~ 27 1 [SING~ 2 [NO] 3 [COMP~ 1 [WOR~ 1 [YES]    100
## 7  184 2 [FEMA~ 34 4 [MARR~ 1 [YES] 5 [COMP~ 5 [FAM~ 2 [NO]      0
## 8  183 1 [MALE~ 35 1 [SING~ 2 [NO] 4 [SOME~ 7 [LIF~ 2 [NO]      0
## 9  144 2 [FEMA~ 43 4 [MARR~ 1 [YES] 2 [SOME~ 2 [SPO~ 2 [NO]      NA
## 10  57 1 [MALE~ 50 4 [MARR~ 1 [YES] 4 [SOME~ 1 [WOR~ 2 [NO]      0
## # ... with 424 more rows
```

```
read_spss("survey.sav", n_max = 10)
```

```
## # A tibble: 434 x 9
##   id      sex age marital  child   educ  source  smoke smokenum
##   <dbl> <dbl+lb> <dbl> <dbl+lb> <dbl+1> <dbl+lb> <dbl+lb> <dbl+1> <dbl>
## 1  484 2 [FEMA~ 31 4 [MARR~ 1 [YES] 5 [COMP~ 7 [LIF~ 2 [NO]      NA
## 2  341 2 [FEMA~ 30 6 [SEPA~ 2 [NO] 4 [SOME~ 8 [MON~ 2 [NO]      0
## 3  300 1 [MALE~ 23 2 [STEA~ 2 [NO] 5 [COMP~ 1 [WOR~ 1 [YES]    100
## 4   61 2 [FEMA~ 18 2 [STEA~ 2 [NO] 2 [SOME~ 2 [SPO~ 1 [YES]    40
## 5   24 1 [MALE~ 23 1 [SING~ 2 [NO] 6 [POST~ NA      2 [NO]      0
## 6  138 1 [MALE~ 27 1 [SING~ 2 [NO] 3 [COMP~ 1 [WOR~ 1 [YES]    100
## 7  184 2 [FEMA~ 34 4 [MARR~ 1 [YES] 5 [COMP~ 5 [FAM~ 2 [NO]      0
## 8  183 1 [MALE~ 35 1 [SING~ 2 [NO] 4 [SOME~ 7 [LIF~ 2 [NO]      0
## 9  144 2 [FEMA~ 43 4 [MARR~ 1 [YES] 2 [SOME~ 2 [SPO~ 2 [NO]      NA
## 10  57 1 [MALE~ 50 4 [MARR~ 1 [YES] 4 [SOME~ 1 [WOR~ 2 [NO]      0
## # ... with 424 more rows
```

Além dessas opções, as funções também oferecem o argumento `col_select`, pelo qual você pode definir quais colunas do arquivo devem ser importadas. Esse recurso é particularmente interessante quando você possui um arquivo muito grande, como os microdados da PNAD contínua, e você deseja utilizar apenas algumas colunas, ou apenas algumas variáveis da pesquisa. Para selecionar

colunas no argumento `col_select`, você pode fornecer um vetor contendo os nomes das colunas desejadas, porém, uma outra alternativa mais útil é utilizar um vetor de índices que representam a ordem das colunas desejadas.

```
read_spss("survey_complete.sav", col_select = 45:52)
```

```
## # A tibble: 439 x 8
##   lifsat3 lifsat4 lifsat5 pss1 pss2 pss3 pss4 pss5
##   <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1      5      4      3      3      3      4      3      4
## 2      5      7      5      2      2      3      5      4
## 3      7      6      6      1      2      2      4      4
## 4      7      7      6      4      3      5      5      4
## 5      4      3      3      2      2      3      2      3
## 6      2      2      2      1      1      3      4      3
## 7      1      1      1      4      4      4      2      2
## 8      5      4      6      3      3      5      3      2
## 9      2      1      1      4      4      5      2      1
## 10     1      1      1      4      5      5      1      1
## # ... with 429 more rows
```

```
read_dta("pnad_2015.dta", col_select = c("uf", "v0102", "v0103", "cor", "sexo"))
```

```
## # A tibble: 164,204 x 5
##   uf      v0102 v0103      cor      sexo
##   <dbl>   <dbl> <dbl> <dbl+lbl> <dbl+lbl>
## 1    31 31001718    14 8 [parda] 1 [masculino]
## 2    15 15003760     1 8 [parda] 1 [masculino]
## 3    35 35002425     8 2 [branca] 1 [masculino]
## 4    43 43000126    15 4 [preta] 0 [feminino]
## 5    33 33001812    18 8 [parda] 1 [masculino]
## 6    17 17000440     3 8 [parda] 1 [masculino]
## 7    15 15002683     3 8 [parda] 1 [masculino]
## 8    15 15003639     6 2 [branca] 1 [masculino]
## 9    22 22000194     7 8 [parda] 0 [feminino]
## 10   26 26005808     4 8 [parda] 1 [masculino]
## # ... with 164,194 more rows
```

3.11 *Encoding de caracteres*

Quando nós estamos trabalhando com dados em um computador, estamos lidando com registros digitalizados de informação, e esses registros quase sempre contêm letras e palavras, ou simplesmente, variáveis textuais (*strings* ou caracteres). Dados geográficos, por exemplo, usualmente vem acompanhado de certas informações textuais, como partes de um endereço (cidade, região, rua,

etc.), que dão suporte à identificação e localização de certa informação. Como um outro exemplo, dados de uma pesquisa amostral comumente possuem variáveis qualitativas que funcionam como rótulos, e que categorizam cada pessoa entrevistada em um certo grupo (homem ou mulher; branco, pardo, preto, amarelo ou indígena; etc.).

Em uma escala microscópica, as informações presentes em um computador são armazenadas como *bytes* de informação, que por sua vez são formados por *bits* de informação, que nada mais são do que combinações específicas de 0's e 1's. Com esse fato, eu quero destacar que os nossos computadores não são capazes de guardar diretamente letras, palavras e outros valores textuais. Na verdade, o que os nossos computadores são capazes de guardar, são os códigos binários que em conjunto formam os *bytes* de informação que representam cada uma das letras, ou cada um dos caracteres que formam a sua palavra, o seu parágrafo ou o seu capítulo. Como exemplo, o nome “Belo Horizonte”, é representado em meu computador através da seguinte sequência de *bytes*:

```
charToRaw("Belo Horizonte")
```

```
## [1] 42 65 6c 6f 20 48 6f 72 69 7a 6f 6e 74 65
```

Cada um dos *bytes* acima, representam uma letra, e para que o seu computador seja capaz de relacionar cada um desses *bytes* às respectivas letras que eles representam, ele utiliza um sistema que nós chamamos de *encoding*. É possível que o sistema operacional de seu computador utilize um sistema de *encoding* diferente do meu. Com isso, os *bytes* que representam o nome “Belo Horizonte” em seu computador, podem ser diferentes dos *bytes* acima.

3.11.1 Um pouco sobre fontes, *encoding* e tipografia

Para apresentar visualmente em sua tela, uma palavra ou um texto, o seu computador precisa relacionar caracteres (*characters*) com os seus respectivos *glyphs* (HARALAMBOUS, 2007). Uma fonte que se encontra em seu computador, representa um conjunto de *glyphs*. Um *glyph* é uma imagem ou um desenho de cada letra que está definida dentro dessa fonte. Quando você está, por exemplo, escrevendo um novo documento no Word, e você aperta a tecla “A”, um caractere (que corresponde a letra A) é enviado para o seu computador. Após o seu computador descobrir o *glyph* da fonte que você está utilizando, que corresponde ao caractere A, o Word vai desenhar a palavra A em seu documento, através do *glyph* que corresponde a esse caractere (HARALAMBOUS, 2007).

Ou seja, quando as letras A e A aparecem em sua tela, elas representam o mesmo caractere, mas utilizam diferentes *glyphs* para serem desenhadas na tela de seu computador, pois ambos os caracteres utilizam fontes diferentes. Por um outro ângulo, nós podemos escrever uma frase de mesmo significado em diferentes línguas, porém, muito provavelmente vamos utilizar diferentes caracteres em cada língua. Por exemplo, ao escrevermos “Olá”, “Hello”, “Bonjour” ou “你好”, estamos dizendo a mesma coisa, porém, estamos utilizando caracteres ou letras bem diferentes para tal ato.

Portanto, quando importamos os nossos dados para dentro do R, qualquer informação ou variável textual que esteja presente nesses dados, são guardadas em nosso computador como *bytes* de informação; e o sistema que o nosso computador utiliza, para traduzir esses *bytes* de informação, em

caracteres, que futuramente serão renderizados em nossa tela, através dos *glyphs* que os representa, é chamado de *encoding* (HARALAMBOUS, 2007).

Os primeiros sistemas de *encoding* eram capazes de representar apenas as letras de línguas anglo-saxônicas. Porém, a medida em que os chineses precisavam escrever um relatório em sua língua, ou a partir do momento em que o povo nórdico precisava representar em seus computadores os diferentes acentos presentes em seu alfabeto, diversos outros sistemas de *encoding* foram sendo desenvolvidos ao longo do tempo. Por isso, nós temos hoje uma miscelânea muito grande de sistemas em uso no mundo. Sendo essa confusão, a principal motivação por trás do desenvolvimento do sistema Unicode, que busca universalizar todos esses sistemas em um só (HARALAMBOUS, 2007).

3.11.2 Problemas que emergem do *encoding*

Por que esse assunto é importante dentro da leitura e escrita de arquivos? Porque diferentes arquivos podem utilizar diferentes sistemas de *encoding*, e se quisermos trabalhar corretamente com os dados textuais presentes nesses arquivos, nós devemos interpretá-los através do sistema de *encoding* correto.

Quando você lê um arquivo de acordo com um sistema de *encoding* diferente do sistema que o arquivo de fato utiliza, uma troca de caracteres (ou de letras) ocorre. Com isso, os textos presentes em seu arquivo, ou no nosso caso, em nossos dados, podem ficar bem estapafúrdios. Devido a essa troca de caracteres, há grandes chances de que uma simples pesquisa por algum caractere específico, fique prejudicada.

Como exemplo, eu possuo abaixo um vetor `t` contendo algumas palavras. Ao utilizar a função `grep()` para pesquisar por qualquer palavra que contenha a letra “Á”. Como resultado, a função nos retorna o número 1, indicando que o primeiro elemento do vetor (a palavra “Árabe”) possui essa letra.

```
t <- c("Árabe", "Francês", "Japonês", "Chinês")
```

```
grep("Á", x = t)
```

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

Agora, se eu pedir ao R, que interprete o vetor `t` segundo um *encoding* diferente, perceba que a função `grep()` não é mais capaz de encontrar uma palavra que contenha a letra “Á”.

```
Encoding(t) <- "UTF-8"
```

```
t
```

```
## [1] "<c>rabe" "Franc<ea>s" "Japon<ea>s" "Chin<ea>s"
```

```
grep("Ã", x = t)
```

```
## integer(0)
```

Na hipótese de você abrir um arquivo e estar utilizando o *encoding* incorreto, desde de que você não salve esse arquivo enquanto ele estiver dessa forma, você não irá corromper o seu arquivo. Em resumo, se algum caractere de seu texto não estiver da forma como você esperava, não salve o seu arquivo! Antes, você precisa ajustar o *encoding* de leitura do arquivo, até o momento em que a leitura dos textos presentes em seu arquivo esteja correta.

Apenas para que esse problema fique claro, vamos pegar como exemplo, o arquivo `livros.txt`, que utiliza o sistema de *encoding* UTF-8.

```
livros <- read_csv("livros.txt")
```

```
livros
```

```
# A tibble: 4 x 3
```

	Titulo	Autor	Preco
	<chr>	<chr>	<dbl>
1	O Hobbit	J. R. R. Tolkien	40.7
2	Matemática para Economistas	Carl P. Simon e Lawrence B~	140.
3	Microeconomia: uma Abordagem Mode~	Hal R. Varian	141.
4	A Luneta Âmbar	Philip Pullman	42.9

Agora, veja abaixo o que acontece se utilizarmos o *encoding* errado na leitura do arquivo. Alguns sistemas de *encoding* são relativamente próximos e, por isso, menos trocas tendem a ocorrer em seus textos quando utilizamos o *encoding* errado. Porém, alguns sistemas são muito divergentes e, portanto, os seus textos podem ficar bem bizarros. Perceba abaixo, que ao utilizarmos o *encoding* Latin1, apenas as letras acentuadas foram trocadas.

```
livros <- read_csv("livros.txt", locale = locale(encoding = "Latin1"))
```

```
livros
```

```
# A tibble: 4 x 3
```

	Titulo	Autor	Preco
	<chr>	<chr>	<dbl>
1	"O Hobbit"	J. R. R. Tolkien	40.7
2	"MatemÃítica para Economistas"	Carl P. Simon e Lawrence ~	140.
3	"Microeconomia: uma Abordagem Mode~	Hal R. Varian	141.
4	"A Luneta Ã\u0082mbar"	Philip Pullman	42.9

Portanto, tudo o que precisamos fazer aqui, é voltar para o *encoding* correto de leitura, ao ajustar o valor utilizado no argumento `encoding` de `locale()`, como vimos na seção **Compreendendo o argumento locale**. Em geral, no Brasil se utiliza o sistema ISO-8859-1, ou simplesmente Latin1. Já as funções do pacote `readr` utilizam por padrão, o sistema UTF-8, por isso, você terá de ajustar o *encoding* de leitura com certa frequência.

3.11.3 A função `guess_encoding()` como um possível guia

Nem sempre temos a sorte de sabermos o *encoding* utilizado por um certo arquivo. Por isso, o pacote `readr` oferece a função `guess_encoding()`, que pode descobrir o *encoding* utilizado por certo arquivo. Como foi destacado por Wickham e Grolemund (2017, p. 133), essa função funciona melhor quando você possui uma quantidade grande de texto no qual ela pode se basear. Além disso, ela não é certa 100% do tempo, porém, ela lhe oferece um início razoável caso você esteja perdido.

Para utilizar essa função, você precisa fornecer o seu texto como *bytes*. Ou seja, antes de utilizar essa função, você muito provavelmente terá de converter o seu texto para *bytes*⁴. Para isso, você pode utilizar a função `charToRaw()`, entretanto, essa função busca transformar um vetor de comprimento 1, logo, para utilizarmos essa função, temos de inserir todos os nossos valores textuais em um único *string*.

Como exemplo, vamos utilizar a coluna Municípios do arquivo `Cod_IBGE.txt`, que possui os nomes dos municípios do estado de Minas Gerais. Perceba abaixo, que o arquivo utiliza um *encoding* diferente do padrão utilizado pela função `read_csv2()`, pois a quarta coluna que deveria se chamar Municípios, foi interpretada como `Munic<U+653C><U+3E64>pios`.

```
df <- read_csv2("Cod_IBGE.txt")

-- Column specification -----
cols(
  IBGE = col_double(),
  IBGE2 = col_double(),
  SEF = col_double(),
  `Munic<U+653C><U+3E64>pios` = col_character()
)
```

Para unir todos os nomes de municípios, presentes na coluna `Munic<U+653C><U+3E64>pios`, nós podemos utilizar a função `paste()`, de acordo com as especificações abaixo. Em seguida, podemos transformar o resultado de `paste()` em um vetor de *bytes*, e fornecê-lo para a função `guess_encoding()`.

```
t <- paste(df[[4]], collapse = " ")

raw <- charToRaw(t)

guess_encoding(raw)
```

```
## # A tibble: 2 x 2
##   encoding confidence
##   <chr>          <dbl>
## 1 ISO-8859-1      0.57
## 2 ISO-8859-2      0.27
```

⁴Vetores contendo *bytes* de informação são comumente chamados de *raw vectors* pela comunidade de R.

Repare que a função nos deu 57% de chance do arquivo `Cod_IBGE.txt` estar utilizando o *encoding* ISO-8859-1, que é de fato o *encoding* utilizado pelo arquivo.

Exercícios

Lembre-se que, um arquivo de texto, nada mais é do que um arquivo simples contendo um conjunto de textos. Esses textos são organizados em linhas (onde cada linha representa uma observação diferente), e em cada linha desse arquivo, esses textos são separados em diferentes colunas, através de algum caractere especial, como vírgulas (,), ou pontos e vírgulas (;).

Exercício 1

Como descrevemos ao longo desse capítulo, arquivos de texto são talvez o principal formato de arquivo utilizado hoje para o compartilhamento de dados. Por isso, os próximos exercícios buscam reforçar os conhecimentos a respeito desses arquivos.

- A) Considerando o arquivo de texto contido no objeto `t` abaixo, qual é o caractere especial que define as colunas desse arquivo? Dado que você tenha identificado esse caractere especial, quais comandos você utilizaria para ler esse arquivo?

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

- B) Perceba abaixo, que os objetos `pac1` e `pac2` são praticamente iguais. Perceba também, que estamos utilizando os mesmos comandos de importação para ambos os objetos. Porém, os resultados gerados pela função são diferentes em cada objeto. Tente identificar o que está causando essa diferença. Dado que você tenha identificado a fonte de tal diferença, como você ajustaria os comandos aplicados sobre cada objeto, de forma que os seus resultados sejam exatamente iguais?

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

```
pac2 <- "Setor;Produção;Receita;Gasto em P&D
Produtos alimentícios;10.828,37;199907,55;3358,36"
```

```

Bebidas;759,53;28093,21;x
Produtos do fumo;69,99;8863,5;121,35
Produtos têxteis;4.153,97;25804,16;746,83
Produtos de madeira;5.088,78;15320,69;279,54
Celulose e outras pastas;26,95;4245,19;216,7
Refino de petróleo;75,48;114316,31;1550,73
Produtos químicos;3.179,52;133582,8;2914,09
Produtos farmacêuticos;621,82;24972,07;1038,73"

```

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

```
## # A tibble: 9 x 4
```

##	Setor	Produção	Receita	`Gasto em P&D`
##	<chr>	<dbl>	<dbl>	<dbl>
## 1	Produtos alimentícios	1082837	19990755	335836
## 2	Bebidas	75953	2809321	NA
## 3	Produtos do fumo	6999	88635	12135
## 4	Produtos têxteis	415397	2580416	74683
## 5	Produtos de madeira	508878	1532069	27954
## 6	Celulose e outras pastas	2695	424519	2167
## 7	Refino de petróleo	7548	11431631	155073
## 8	Produtos químicos	317952	1335828	291409
## 9	Produtos farmacêuticos	62182	2497207	103873

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

```
## # A tibble: 9 x 4
```

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

- c) Considerando que você tenha chamado com sucesso pelo pacote readr, com o comando library(), você será capaz de executar os comandos mostrados abaixo sem problemas. Tais comandos buscam importar para o R, um arquivo chamado challenge.csv (a função readr_example() nos traz a localização desse arquivo challenge.csv em seu computador). Porém, perceba pelo resultado abaixo, que erros de importação ocorreram em 1000 linhas do arquivo.

```
import <- read_csv(readr_example("challenge.csv"))
```

```
## -- Column specification -----
## cols(
##   x = col_double(),
##   y = col_logical()
## )

## Warning: 1000 parsing failures.
## row col          expected    actual          file
## 1001  y 1/0/T/F/TRUE/FALSE 2015-01-16 'C:/Users/Pedro/Documen~
## 1002  y 1/0/T/F/TRUE/FALSE 2018-05-18 'C:/Users/Pedro/Documen~
## 1003  y 1/0/T/F/TRUE/FALSE 2015-09-05 'C:/Users/Pedro/Documen~
## 1004  y 1/0/T/F/TRUE/FALSE 2012-11-28 'C:/Users/Pedro/Documen~
## 1005  y 1/0/T/F/TRUE/FALSE 2020-01-13 'C:/Users/Pedro/Documen~
## ....
## See problems(...) for more details.
```

Ao navegar por todo o conteúdo desse arquivo `challenge.csv`, você pode perceber que os dados contidos nesse arquivo foram incorretamente interpretados pela função `read_csv()`. Com isso, o seu trabalho será descobrir o que deu errado nesse processo, e ajustar os comandos de importação desse arquivo para que esse erro não ocorra.

Como uma dica, abra o arquivo `readr_example("challenge.csv")` e veja o seu conteúdo com cuidado. Com os comandos abaixo, você pode navegar por esse arquivo em uma janela de seu próprio RStudio. Portanto, tente descobrir o que está acontecendo de errado, e crie um comando que possa corrigir esse problema de importação.

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

- D) Considerando o objeto `t` abaixo, como você faria para importar corretamente esse arquivo? Vale ressaltar, que temos uma coluna contendo datas dentro do objeto `t`, e, até o momento, nós ainda não discutimos como o R interpreta ou lida com esse tipo de variável. Tal discussão é feita no capítulo 12 ([Introdução à variáveis de tempo com lubridate](#)). Portanto, não se preocupe caso você não consiga importar especificamente essa coluna da maneira correta. De qualquer maneira, ao final desse livro, nós fornecemos todo o código necessário para interpretar corretamente essa coluna.

```
t <- "Data_execução*Unidades*Valor_compra
20/01/2020*21*R$ 3049,50
23/01/2020*502*R$ 1289,03
25/01/2020*90*R$ 678,00
02/02/2020*123*R$ 5401
05/02/2020*45*R$ 1450,10
07/02/2020*67*R$ 2320,97
09/02/2020*187*R$ 6231,76"
```

Exercício 2

Copie e cole o endereço URL abaixo em seu navegador de preferência. Com esse link, uma planilha em Excel será baixada. Nessa planilha, temos alguns dados referentes aos municípios de Minas Gerais, ou, mais especificamente, a como esses municípios se encaixam no critério de Produção de Alimentos no âmbito da lei estadual 18.030/2009. Tente criar um comando que possa importar corretamente os dados dessa planilha para o R.

- A) https://github.com/pedropark99/Curso-R/blob/master/Dados/emater_icms_solidario.xlsx?raw=true

Capítulo 4

Transformando dados com dplyr

4.1 Introdução e pré-requisitos

São raras as ocasiões em que os seus dados já se encontram no formato exato que você precisa para realizar as suas análises, ou para gerar os gráficos que você deseja ([WICKHAM; GROLEMUND, 2017](#)). Por essa razão, você irá passar uma parte considerável de seu tempo, aplicando transformações sobre os seus dados, e calculando novas variáveis, e como os seus dados estarão, na maioria das situações, alocados em um `data.frame`, você precisa de ferramentas que sejam eficientes com tal estrutura ([PENG, 2015](#)). Esse é o objetivo do pacote `dplyr` com o qual vamos trabalhar neste capítulo.

Para que você tenha acesso as funções e possa acompanhar os exemplos desse capítulo, você precisa chamar pelo pacote `dplyr`. Porém, vamos utilizar algumas bases de dados que estão disponíveis através de outros pacotes presentes no `tidyverse`. Por isso, é preferível que você chame pelo `tidyverse` por meio do comando `library()`.

```
library(tidyverse)
library(dplyr)
```

4.2 Panorama e padrões do pacote dplyr

Segundo a [página oficial](#), o pacote `dplyr` busca oferecer um conjunto de “verbos” (i.e. funções) voltados para as operações mais comumente aplicadas em tabelas. Ou seja, as funções desse pacote em geral aceitam um `data.frame` como *input*, e retornam um novo `data.frame` como *output*.

Se você possui alguma experiência com *relational database management systems* (RDBMS), você vai acabar percebendo ao longo deste capítulo, que o pacote `dplyr` é profundamente inspirado nos verbos da linguagem SQL. Por esse motivo, muitos usuários tendem a chamar as funções do pacote `dplyr` de “verbos”. Logo abaixo, temos uma lista das principais funções oferecidas pelo pacote, além de uma descrição rápida da ação realizada por cada um desses verbos.

- 1) `select()`: busca selecionar ou extrair colunas de seu `data.frame`.
- 2) `filter()`: busca filtrar linhas de seu `data.frame`.
- 3) `arrange()`: busca ordenar (ou organizar) as linhas de seu `data.frame`.
- 4) `mutate()`: busca adicionar ou calcular novas colunas em seu `data.frame`.
- 5) `summarise()`: busca sintetizar múltiplos valores de seu `data.frame` em um único valor.
- 6) `group_by()`: permite que as operações sejam executadas dentro de cada “grupo” de seu `data.frame`; em outras palavras, a função busca definir os grupos existentes em seu `data.frame`, e deixar essa definição explícita e disponível para os outros verbos, de modo que eles possam respeitar esses grupos em suas operações.

Dentre os verbos acima, o `group_by()` é definitivamente o mais difícil de se explicar de uma maneira clara e, ao mesmo tempo, resumida. De qualquer maneira, vamos discutir ele por extenso na seção **Agrupando dados e gerando estatísticas sumárias com `group_by()` e `summarise()`**. Além disso, também vamos abordar nesse capítulo, o uso do operador *pipe* (%>%) que provém do pacote *magrittr*, e que hoje, faz parte da identidade do pacote *dplyr*, e do *tidyverse* como um todo.

Peng (2015) destacou algumas características compartilhadas pelas funções do pacote *dplyr*:

- 1) Possuem como primeiro argumento (`.data`), o `data.frame` no qual você deseja aplicar a função que você está utilizando.
- 2) Os argumentos subsequentes buscam descrever como e onde aplicar a função sobre o `data.frame` definido no primeiro argumento.
- 3) Geram um novo `data.frame` como resultado.
- 4) Como você definiu o `data.frame` a ser utilizando no primeiro argumento da função, você pode se referir às colunas desse `data.frame` apenas pelo seus nomes. Ou seja, dentro das funções do pacote *dplyr*, você não precisa mais do operador `$` para acessar as colunas do `data.frame` utilizado.

No momento, essas características podem parecer difusas. Porém, você irá rapidamente reconhecê-las ao longo deste capítulo.

4.3 Operador *pipe* (%>%)

Hoje, o operador *pipe* (%>%) faz parte da identidade dos pacotes que compõem o *tidyverse* e, por isso, você irá encontrar esse operador em praticamente qualquer *script* que utilize algum desses pacotes. Grande parte dessa identidade foi construída nos últimos anos, em especial, com a obra de Wickham e Grolemund (2017) que se tornou um importante livro-texto da linguagem R como um todo.

O operador *pipe* provém do pacote *magrittr*, e o seu único objetivo é tornar o seu código mais claro e compreensível. Ou seja, o *pipe* em nada altera o resultado ou as configurações de seus

comandos, ele apenas os organiza em uma estrutura mais limpa e arranjada. Apesar de sua origem ser o pacote *magrittr*, o *pipe* é carregado automaticamente quando chamamos pelo *tidyverse*, através do comando `library()`. Com isso, temos duas opções para termos acesso a esse operador: chamar pelo pacote *magrittr*, ou chamar pelo *tidyverse*.

```
## Com um desses comandos você  
## pode utilizar o operador %>%  
library(tidyverse)  
## Ou  
library(magrittr)
```

ATALHO: No RStudio, você pode criar um *pipe* através do atalho Ctrl + Shift + M.

Mesmo que esse efeito seja simples, ele é extremamente importante. A estrutura em “cadeia” construída pelo *pipe* gera uma grande economia em seu tempo de trabalho, pois você não precisa mais se preocupar em salvar o resultado de vários passos intermediários em algum objeto. Dessa maneira, você pode focar mais tempo nas próprias transformações em si, e no resultado que você deseja atingir. Além disso, essa estrutura também vai salvar muito de seu tempo, nos momentos em que você retornar ao seu trabalho no dia seguinte. Pois se o seu código nessa estrutura está mais claro e fácil de se ler, você pode recuperar com maior rapidez a compreensão do ponto em que você parou no dia anterior.

Isso é muito importante, pois você nunca está trabalhando sozinho! Você sempre está, no mínimo, trabalhando com o seu futuro eu (WICKHAM; GROLEMUND, 2017). Por isso, qualquer quantidade de tempo que você emprega para tornar os seus comandos mais legíveis e eficientes, você estará automaticamente economizando o seu tempo no dia seguinte, quando você terá de retornar a esses comandos, e prosseguir com o seu trabalho. Para mais, os seus possíveis colegas de trabalho, ou outras pessoas que estiverem envolvidas no desenvolvimento de seu *script*, vão compreender de maneira mais eficiente as transformações que você está aplicando e, portanto, vão ser capazes de contribuir com o seu trabalho de maneira mais rápida.

4.3.1 O que o *pipe* faz ?

Em qualquer análise, temos em geral diversas etapas ou transformações a serem executadas, e em sua maioria, essas etapas assumem uma ordem específica. Quando realizamos essas etapas no R, nós comumente salvamos os resultados de cada passo em novos objetos, e utilizamos esses objetos “intermediários” em cada operação adicional para chegarmos ao resultado final que desejamos. Perceba no exemplo abaixo, o trabalho que temos ao salvarmos os resultados de cada passo em um objeto, e utilizarmos esse objeto na próxima transformação.

```
dados <- mpg  
agrupamento <- group_by(.data = dados, class)  
base_ordenada <- arrange(.data = agrupamento, hwy)  
base_completa <- mutate(
```

```
.data = base_ordenada,
media = mean(hwy),
desvio = hwy - media
)
```

Aqui se encontra uma vantagem importante do operador *pipe*, pois ele elimina essa necessidade de objetos “intermediários”, ao “carregar” os resultados ao longo de diversas funções. Em outras palavras, esse operador funciona como uma ponte entre cada etapa, ou entre cada função aplicada. Dito de uma maneira mais específica, quando conectamos duas funções por um *pipe*, o operador carrega o resultado da primeira função, e o insere como o primeiro argumento da segunda função. Com isso, eu posso reescrever os comandos anteriores da seguinte forma:

```
mpg %>%
  group_by(class) %>%
  arrange(hwy) %>%
  mutate(
    media = mean(hwy),
    desvio = hwy - media
  )
```

Além das vantagens destacadas até o momento, ao evitar o uso de objetos “intermediários”, o *pipe* acaba evitando que você use desnecessariamente a memória de seu computador. Pois cada objeto criado no R, precisa ocupar um espaço de sua memória RAM para permanecer “vivo” e disponível em sua sessão. Como evitamos a criação desses objetos “intermediários”, estamos utilizando menos memória para realizar exatamente as mesmas etapas e gerar os mesmos resultados.

Apenas para que o uso do *pipe* fique claro, se eu possuo as funções *x()*, *y()* e *z()*, e desejo calcular a expressão *z(y(x(10), times = 1), n = 20, replace = TRUE)*, nós podemos reescrever essa expressão do modo exposto abaixo. Dessa maneira, o *pipe* vai pegar o resultado de *x(10)*, e inseri-lo como o primeiro argumento da função *y()*; depois de calcular o resultado da função *y()*, o próximo *pipe* vai passá-lo para a função *z()*; e como a função *z()* é a última função da cadeia, o console vai lhe mostrar o resultado final desse processo.

```
## Expressão original
z(y(x(10), times = 1), n = 20, replace = TRUE)

## Com o uso do pipe %>%
x(10) %>%
  y(times = 1) %>%
  z(n = 20, replace = TRUE)
```

4.3.2 O que o *pipe* não é capaz de fazer ?

O *pipe* não é capaz de trabalhar perfeitamente com qualquer função, e a principal característica que você precisa observar para identificar se essa afirmação é verdadeira ou não para uma dada função,

é o seu primeiro argumento.

Como o *pipe* insere o **resultado** da expressão anterior no primeiro argumento da próxima função, esse primeiro argumento precisa corresponder ao argumento no qual você deseja utilizar esse **resultado**. Na maior parte do tempo, desejamos utilizar esse **resultado** como os dados sobre os quais vamos aplicar a nossa próxima função. Este é um dos principais motivos pelos quais praticamente todas as funções de todos os pacotes que compõe o tidyverse, trabalham perfeitamente bem com o operador *pipe*. Pois todas essas funções possuem como primeiro argumento, algo parecido com `.data`, `data` ou `x`, que busca definir o objeto sobre o qual vamos aplicar a função.

Caso o argumento a ser utilizado, esteja em uma posição diferente (se trata do segundo, terceiro ou quarto argumento da função), você pode utilizar um ponto final (`.`) para alterar a posição em que o resultado das etapas anteriores será introduzido. Basta posicionar o ponto final no argumento em que você deseja inserir esse resultado.

Um clássico exemplo que se encaixa nessa hipótese, é a função `lm()`, que é a principal função empregada no cálculo de uma regressão linear no R. Nessa função, o primeiro argumento corresponde a fórmula a ser utilizada na regressão; já os dados a serem usados na regressão, são delimitados no segundo argumento da função (`data`). Veja no exemplo abaixo, que eu utilizo um ponto final sobre o argumento `data`, para dizer ao *pipe* que ele deve inserir o resultado anterior especificamente nesse argumento.

```
mpg %>%
  lm(hwy ~ cyl, data = .) %>%
  summary()

##
## Call:
## lm(formula = hwy ~ cyl, data = .)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -8.7579 -2.4968  0.2421  2.4379 15.2421
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  40.0190     0.9591   41.72  <2e-16 ***
## cyl         -2.8153     0.1571  -17.92  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.865 on 232 degrees of freedom
## Multiple R-squared:  0.5805, Adjusted R-squared:  0.5787
## F-statistic: 321.1 on 1 and 232 DF,  p-value: < 2.2e-16
```

4.3.3 Duas dicas rápidas sobre o *pipe*

O *pipe* cria uma espécie de efeito em cadeia, e muitas vezes nos preocupamos demais com as etapas dessa cadeia, e nos esquecemos de definir o local em que o resultado dessa cadeia deve ocupar. Portanto, lembre-se que para salvar o resultado final da cadeia formada pelos seus *pipe*'s, você necessita salvar esse resultado em algum objeto. Para isso, você deve posicionar o nome do objeto, e o símbolo de *assignment* (<-), logo no início dessa cadeia, como no exemplo abaixo.

```
resultado <- mpg %>%  
  group_by(class) %>%  
  arrange(hwy) %>%  
  mutate(  
    media = mean(hwy),  
    desvio = hwy - media  
  )
```

Uma outra dica, seria não formar cadeias muito longas. Como um guia, uma cadeia de *pipe*'s não deveria passar de 7 etapas. Caso você precise aplicar mais do que 7 etapas, é melhor que você salve o resultado da 7ª etapa em um objeto, e inicie uma nova cadeia a partir deste objeto.

4.4 Selecionando colunas com select()

Como definimos anteriormente, a função `select()` busca selecionar colunas de seu `data.frame`. Você já possui uma boa ideia de como realizar essa ação através da função de *subsetting* (`[]`). Porém, nós podemos usufruir da flexibilidade oferecida pela função `select()`, que lhe permite realizar essa mesma operação de diversas maneiras intuitivas.

No geral, temos ao menos 5 métodos diferentes que podemos utilizar na função `select()`:

- 1) simplesmente listar o nome das colunas que desejamos;
- 2) fornecer um vetor externo, contendo os nomes das colunas a serem extraídas;
- 3) selecionar um conjunto de colunas com base em seu tipo (`integer`, `double`, `character`, `logical`);
- 4) selecionar um conjunto de colunas com base em padrões que aparecem nos nomes dessas colunas (nome começa por y, ou termina em z, ou contém x);
- 5) selecionar um conjunto de colunas com base em seus índices numéricos (1ª colunas, 2ª coluna, 3ª coluna, etc.).

Como exemplo inicial, vamos utilizar a tabela `billboard`, que apresenta a posição de diversas músicas na lista Billboard Top 100, ao longo do ano de 2000. Se você chamou com sucesso pelo `tidyverse`, você tem acesso a essa tabela. Perceba que a posição de cada música descrita na tabela, é apresentada de forma semanal, onde cada semana possui a sua coluna própria. Por essa razão, temos uma quantidade exorbitante de colunas na tabela.

billboard

```
## # A tibble: 317 x 79
##   artist track date.entered wk1 wk2 wk3 wk4 wk5 wk6 wk7
##   <chr> <chr> <date> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2 Pac Baby Do~ 2000-02-26 87 82 72 77 87 94 99
## 2 2Ge+her The Har~ 2000-09-02 91 87 92 NA NA NA NA
## 3 3 Doors~ Krypton~ 2000-04-08 81 70 68 67 66 57 54
## 4 3 Doors~ Loser 2000-10-21 76 76 72 69 67 65 55
## 5 504 Boyz Wobble ~ 2000-04-15 57 34 25 17 17 31 36
## 6 98^0 Give Me~ 2000-08-19 51 39 34 26 26 19 2
## 7 A*Teens Dancing~ 2000-07-08 97 97 96 95 100 NA NA
## 8 Aaliyah I Don't~ 2000-01-29 84 62 51 41 38 35 35
## 9 Aaliyah Try Aga~ 2000-03-18 59 53 38 28 21 18 16
## 10 Adams, ~ Open My~ 2000-08-26 76 76 74 69 68 67 61
## # ... with 307 more rows, and 69 more variables: wk8 <dbl>, wk9 <dbl>,
## # wk10 <dbl>, wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, wk14 <dbl>,
## # wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>,
## # wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
## # wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>,
## # wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,
## # wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>,
## # wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>,
## # wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>,
## # wk50 <dbl>, wk51 <dbl>, wk52 <dbl>, wk53 <dbl>, wk54 <dbl>,
## # wk55 <dbl>, wk56 <dbl>, wk57 <dbl>, wk58 <dbl>, wk59 <dbl>,
## # wk60 <dbl>, wk61 <dbl>, wk62 <dbl>, wk63 <dbl>, wk64 <dbl>,
## # wk65 <dbl>, wk66 <lgl>, wk67 <lgl>, wk68 <lgl>, wk69 <lgl>,
## # wk70 <lgl>, wk71 <lgl>, wk72 <lgl>, wk73 <lgl>, wk74 <lgl>,
## # wk75 <lgl>, wk76 <lgl>
```

O método 5 citado acima é um dos métodos mais práticos e eficientes de se utilizar a função `select()`. Por exemplo, se desejássemos extrair todas as colunas entre a 1º e 4º colunas da tabela, poderíamos fornecer um vetor à função, contendo uma sequência de 1 a 4, que representa os índices das colunas que desejamos, como no exemplo abaixo.

```
billboard_sel <- select(billboard, 1:4)
```

billboard_sel

```
## # A tibble: 317 x 4
##   artist track date.entered wk1
##   <chr> <chr> <date> <dbl>
## 1 2 Pac Baby Don't Cry (Keep... 2000-02-26 87
## 2 2Ge+her The Hardest Part Of ... 2000-09-02 91
```

```
## 3 3 Doors Down Kryptonite 2000-04-08 81
## 4 3 Doors Down Loser 2000-10-21 76
## 5 504 Boyz Wobble Wobble 2000-04-15 57
## 6 98^0 Give Me Just One Nig... 2000-08-19 51
## 7 A*Teens Dancing Queen 2000-07-08 97
## 8 Aaliyah I Don't Wanna 2000-01-29 84
## 9 Aaliyah Try Again 2000-03-18 59
## 10 Adams, Yolanda Open My Heart 2000-08-26 76
## # ... with 307 more rows
```

Agora, e se você precisasse selecionar todas as colunas que representam as semanas? Nesse caso, o método 5 ainda seria uma boa alternativa, pois você precisaria apenas fornecer uma sequência que represente a posição dessas colunas na tabela (de 4 a 79 para ser mais preciso).

Porém, todas essas colunas possuem um padrão em seus nomes. Elas se iniciam pelos caracteres "wk", acrescidos de um número que representa o índice da semana que essa coluna corresponde. Portanto, em todas as ocasiões que houver algum padrão presente nos nomes das colunas que você deseja selecionar, o método 4 que citamos configura-se como uma ótima solução. Nesse método, devemos utilizar as funções de suporte `starts_with()`, `ends_with()`, `matches()`.

Como os seus próprios nomes dão a entender, as funções `starts_with()` e `ends_with()` vão selecionar qualquer coluna de sua tabela que comece (*start*) ou termine (*end*) por uma determinada cadeia de caracteres, respectivamente. Como exemplo, eu posso selecionar todas as colunas que apresentam as posições semanais na tabela `billboard`, ao encontrar todas as colunas que começam pelas letras "wk", com a função `starts_with()`.

```
billboard_sel <- select(billboard, starts_with("wk"))
```

```
billboard_sel
```

```
## # A tibble: 317 x 76
##   wk1  wk2  wk3  wk4  wk5  wk6  wk7  wk8  wk9  wk10 wk11 wk12
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    87    82    72    77    87    94    99    NA    NA    NA    NA    NA
## 2    91    87    92    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 3    81    70    68    67    66    57    54    53    51    51    51    51
## 4    76    76    72    69    67    65    55    59    62    61    61    59
## 5    57    34    25    17    17    31    36    49    53    57    64    70
## 6    51    39    34    26    26    19    2     2     3     6     7    22
## 7    97    97    96    95   100    NA    NA    NA    NA    NA    NA    NA
## 8    84    62    51    41    38    35    35    38    38    36    37    37
## 9    59    53    38    28    21    18    16    14    12    10     9     8
## 10   76    76    74    69    68    67    61    58    57    59    66    68
## # ... with 307 more rows, and 64 more variables: wk13 <dbl>, wk14 <dbl>,
## #   wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>,
```



```
## # wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
## # wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>,
## # wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,
## # wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>,
## # wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>,
## # wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>,
## # wk50 <dbl>, wk51 <dbl>, wk52 <dbl>, wk53 <dbl>, wk54 <dbl>,
## # wk55 <dbl>, wk56 <dbl>, wk57 <dbl>, wk58 <dbl>, wk59 <dbl>,
## # wk60 <dbl>, wk61 <dbl>, wk62 <dbl>, wk63 <dbl>, wk64 <dbl>,
## # wk65 <dbl>, wk66 <lg1>, wk67 <lg1>, wk68 <lg1>, wk69 <lg1>,
## # wk70 <lg1>, wk71 <lg1>, wk72 <lg1>, wk73 <lg1>, wk74 <lg1>,
## # wk75 <lg1>, wk76 <lg1>
```

Já a função `matches()` se trata de um caso muito mais flexível das funções `starts_with()` e `ends_with()`, pois ela lhe permite selecionar qualquer coluna cujo o nome se encaixa em uma dada expressão regular. Expressões regulares são uma poderosa ferramenta para processamento de texto, a qual vamos discutir no capítulo 11 deste livro, especialmente na seção [Expressões regulares \(ou regex\) com `str\_detect\(\)`](#). Outras duas referências úteis sobre o assunto, se encontram no [capítulo 14](#) de Wickham e Grolemund (2017), que provê uma visão mais direta, além da obra de Friedl (2006) que oferece uma visão técnica e aprofundada sobre o assunto. Veja alguns exemplos abaixo:

```
## Selecciona todas as semanas que são
## maiores do que 9 e menores do que 100.
## Ou seja, toda semana com dois dígitos
billboard %>%
  select(matches("wk[0-9]{2}")) %>% print(n = 5)

## # A tibble: 317 x 67
##   wk10 wk11 wk12 wk13 wk14 wk15 wk16 wk17 wk18 wk19 wk20 wk21
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 2    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
## 3    51    51    51    47    44    38    28    22    18    18    14    12
## 4    61    61    59    61    66    72    76    75    67    73    70    NA
## 5    57    64    70    75    76    78    85    92    96    NA    NA    NA
## # ... with 312 more rows, and 55 more variables: wk22 <dbl>, wk23 <dbl>,
## # wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>,
## # wk29 <dbl>, wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>,
## # wk34 <dbl>, wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>,
## # wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>,
## # wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>,
## # wk49 <dbl>, wk50 <dbl>, wk51 <dbl>, wk52 <dbl>, wk53 <dbl>,
## # wk54 <dbl>, wk55 <dbl>, wk56 <dbl>, wk57 <dbl>, wk58 <dbl>,
## # wk59 <dbl>, wk60 <dbl>, wk61 <dbl>, wk62 <dbl>, wk63 <dbl>,
```

```
## #   wk64 <dbl>, wk65 <dbl>, wk66 <lgl>, wk67 <lgl>, wk68 <lgl>,
## #   wk69 <lgl>, wk70 <lgl>, wk71 <lgl>, wk72 <lgl>, wk73 <lgl>,
## #   wk74 <lgl>, wk75 <lgl>, wk76 <lgl>
```

```
## Selecciona todas as colunas cujo nome
## possua um ponto final antecedido por
## 4 letras
billboard %>%
  select(matches("[a-z]{4}[.]")) %>% print(n = 5)
```

```
## # A tibble: 317 x 1
##   date.entered
##   <date>
## 1 2000-02-26
## 2 2000-09-02
## 3 2000-04-08
## 4 2000-10-21
## 5 2000-04-15
## # ... with 312 more rows
```

Essas são maneiras eficientes de selecionarmos um grande conjunto de colunas, porém, muitas vezes as nossas necessidades são pequenas e, portanto, não exigem mecanismos tão poderosos. Nessas situações, o método 1 se torna útil pois ele consiste em simplesmente listarmos o nome das colunas desejadas. Como exemplo, eu posso selecionar as colunas artist, track e wk5 da tabela billboard pelo comando abaixo.

```
billboard %>% select(artist, track, wk5)
```

```
## # A tibble: 317 x 3
##   artist      track      wk5
##   <chr>      <chr>      <dbl>
## 1 2 Pac      Baby Don't Cry (Keep...  87
## 2 2Ge+her    The Hardest Part Of ...  NA
## 3 3 Doors Down Kryptonite             66
## 4 3 Doors Down Loser                  67
## 5 504 Boyz    Wobble Wobble           17
## 6 98^0       Give Me Just One Nig...  26
## 7 A*Teens    Dancing Queen          100
## 8 Aaliyah     I Don't Wanna           38
## 9 Aaliyah     Try Again               21
## 10 Adams, Yolanda Open My Heart  68
## # ... with 307 more rows
```

Vale destacar que a ordem dos índices utilizados importa para a função select(). Logo, se no exemplo acima, eu listasse as colunas na ordem track, wk5 e artist, o novo data.frame resultante

de `select()`, iria conter essas colunas precisamente nessa ordem. O mesmo efeito seria produzido, caso eu utilizasse novamente o método 5, e fornecesse o vetor `c(3, 2, 4)` à função. Dessa forma, `select()` iria me retornar um novo `data.frame` contendo 3 colunas, que correspondem a 3º, 2º e 4º colunas da tabela `billboard`, exatamente nessa ordem.

Por outro lado, não há uma maneira de variarmos a ordem dos resultados gerados nos métodos 3 e 4, especificamente. Por isso, caso você utilize um desses dois métodos, as colunas selecionadas serão apresentadas no novo `data.frame`, precisamente na ordem em que eles aparecem no `data.frame` inicial.

Visto esses pontos, ao invés de selecionar colunas, você também pode utilizar o método 1 para rapidamente eliminar algumas colunas de seu `data.frame`, ao posicionar um sinal negativo (-) antes do nome da coluna que você deseja retirar. Por exemplo, eu posso selecionar todas as colunas da tabela `mpg`, exceto as colunas `hwy` e `manufacturer` por meio do seguinte comando:

```
mpg %>% select(-hwy, -manufacturer)
```

```
## # A tibble: 234 x 9
##   model      displ  year   cyl trans      drv    cty fl   class
##   <chr>      <dbl> <int> <int> <chr>    <chr> <int> <chr> <chr>
## 1 a4          1.8  1999     4 auto(l5) f      18 p   compact
## 2 a4          1.8  1999     4 manual(m5) f      21 p   compact
## 3 a4          2    2008     4 manual(m6) f      20 p   compact
## 4 a4          2    2008     4 auto(av) f      21 p   compact
## 5 a4          2.8  1999     6 auto(l5) f      16 p   compact
## 6 a4          2.8  1999     6 manual(m5) f      18 p   compact
## 7 a4          3.1  2008     6 auto(av) f      18 p   compact
## 8 a4 quattro  1.8  1999     4 manual(m5) 4      18 p   compact
## 9 a4 quattro  1.8  1999     4 auto(l5) 4      16 p   compact
## 10 a4 quattro  2    2008     4 manual(m6) 4      20 p   compact
## # ... with 224 more rows
```

Em contrapartida, o método 3 busca selecionar um conjunto de colunas com base em seu tipo de dado, através da função `where()` e das funções de teste lógico `is.*()` (`is.double`, `is.character`, `is.integer`, ...). Como exemplo, nós podemos selecionar todas as colunas da tabela `billboard` que contém dados textuais, através do comando abaixo. Portanto, para utilizar esse método você precisa apenas se referir a função `is.*()` que corresponde ao tipo de dado no qual você está interessado, dentro da função `where()`.

```
billboard %>% select(where(is.character))
```

```
## # A tibble: 317 x 2
##   artist      track
##   <chr>      <chr>
## 1 2 Pac      Baby Don't Cry (Keep...
```

```
## 2 2Ge+her      The Hardest Part Of ...
## 3 3 Doors Down Kryptonite
## 4 3 Doors Down Loser
## 5 504 Boyz     Wobble Wobble
## 6 98^0         Give Me Just One Nig...
## 7 A*Teens      Dancing Queen
## 8 Aaliyah      I Don't Wanna
## 9 Aaliyah      Try Again
## 10 Adams, Yolanda Open My Heart
## # ... with 307 more rows
```

Com isso, você possui não apenas uma boa variedade de métodos disponíveis na função `select()`, mas você também é capaz de misturá-los livremente dentro da função. Ou seja, se for de meu desejo, eu posso utilizar os métodos 2, 4 e 5 ao mesmo tempo, como no exemplo abaixo. Tratando especificamente do método 2, eu preciso fornecer dentro da função `all_of()`, um vetor contendo os nomes das colunas desejadas. Como exemplo, eu posso novamente extrair as colunas `artist`, `track` e `wk5` através desse método. O método 2, em particular, se torna um método interessante quando ainda não conhecemos o conjunto de colunas a serem extraídas. Talvez você precise aplicar previamente diversos testes sobre o seu `data.frame`, para identificar essas colunas. Logo, um vetor contendo os nomes das colunas desejadas seria o resultado ideal para tais testes.

```
vec <- c("artist", "track", "wk5")

billboard %>% select(
  all_of(vec), ## Método 2
  3:5, ## Método 5
  matches("wk[0-9]{2}") ## Método 4
)

## # A tibble: 317 x 73
##   artist track wk5 date.entered wk1 wk2 wk10 wk11 wk12 wk13
##   <chr> <chr> <dbl> <date> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2 Pac Baby Do~ 87 2000-02-26 87 82 NA NA NA NA
## 2 2Ge+her The Har~ NA 2000-09-02 91 87 NA NA NA NA
## 3 3 Doors~ Krypton~ 66 2000-04-08 81 70 51 51 51 47
## 4 3 Doors~ Loser 67 2000-10-21 76 76 61 61 59 61
## 5 504 Boyz Wobble ~ 17 2000-04-15 57 34 57 64 70 75
## 6 98^0 Give Me~ 26 2000-08-19 51 39 6 7 22 29
## 7 A*Teens Dancing~ 100 2000-07-08 97 97 NA NA NA NA
## 8 Aaliyah I Don't~ 38 2000-01-29 84 62 36 37 37 38
## 9 Aaliyah Try Aga~ 21 2000-03-18 59 53 10 9 8 6
## 10 Adams, ~ Open My~ 68 2000-08-26 76 76 59 66 68 61
## # ... with 307 more rows, and 63 more variables: wk14 <dbl>, wk15 <dbl>,
## # wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>, wk20 <dbl>,
## # wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>, wk25 <dbl>,
```

Tabela 4.1: Lista de operadores lógicos

Operador	Estrutura do teste	Descrição do teste
<	<code>x < y</code>	x menor do que y
>	<code>x > y</code>	x maior do que y
<=	<code>x <= y</code>	x menor ou igual a y
>=	<code>x >= y</code>	x maior ou igual a y
==	<code>x == y</code>	x igual a y
!=	<code>x != y</code>	x não é igual a y
!	<code>!x</code>	não se encaixa na condição definida em x
	<code>x y</code>	se encaixa na condição x ou na condição y
&	<code>x & y</code>	se encaixa na condição x e na condição y
%in%	<code>x %in% y</code>	x está incluso em y

Fonte: Elaboração própria do autor.

```
## # wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
## # wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>,
## # wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>,
## # wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>,
## # wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>, wk50 <dbl>,
## # wk51 <dbl>, wk52 <dbl>, wk53 <dbl>, wk54 <dbl>, wk55 <dbl>,
## # wk56 <dbl>, wk57 <dbl>, wk58 <dbl>, wk59 <dbl>, wk60 <dbl>,
## # wk61 <dbl>, wk62 <dbl>, wk63 <dbl>, wk64 <dbl>, wk65 <dbl>,
## # wk66 <lgl>, wk67 <lgl>, wk68 <lgl>, wk69 <lgl>, wk70 <lgl>,
## # wk71 <lgl>, wk72 <lgl>, wk73 <lgl>, wk74 <lgl>, wk75 <lgl>, wk76 <lgl>
```

4.5 Filtrando linhas com filter()

Você também já possui conhecimento para realizar essa operação através da função *subsetting* (`[]`). Porém, novamente o pacote `dplyr` nos oferece uma alternativa mais intuitiva. A função `filter()` busca filtrar linhas de uma tabela de acordo com uma condição lógica que nós devemos definir. Ou seja, os operadores lógicos são primordiais para essa função. Por isso, temos abaixo na tabela 4.1, um resumo de cada um deles.

Portanto, ao utilizar a função `filter()` você deve construir uma condição lógica que seja capaz de identificar as linhas que você deseja filtrar. Como exemplo inicial, nós podemos retornar à tabela `mpg`, que contém dados de consumo de diversos modelos de carro. Por exemplo, nós podemos filtrar todas as linhas que dizem respeito a modelos da Toyota, através do comando abaixo. Como um paralelo, temos mais abaixo a mesma operação segundo a função de *subsetting*.

```
mpg %>% filter(manufacturer == "toyota")
```

```
## # A tibble: 34 x 11
##   manufacturer model displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 toyota      4run~   2.7  1999     4 manu~ 4     15    20 r    suv
## 2 toyota      4run~   2.7  1999     4 auto~ 4     16    20 r    suv
## 3 toyota      4run~   3.4  1999     6 auto~ 4     15    19 r    suv
## 4 toyota      4run~   3.4  1999     6 manu~ 4     15    17 r    suv
## 5 toyota      4run~   4    2008     6 auto~ 4     16    20 r    suv
## 6 toyota      4run~   4.7  2008     8 auto~ 4     14    17 r    suv
## 7 toyota      camry   2.2  1999     4 manu~ f     21    29 r    mids~
## 8 toyota      camry   2.2  1999     4 auto~ f     21    27 r    mids~
## 9 toyota      camry   2.4  2008     4 manu~ f     21    31 r    mids~
## 10 toyota     camry   2.4  2008     4 auto~ f     21    31 r    mids~
## # ... with 24 more rows
```

```
## -----
```

```
## A mesma operação por subsetting:
```

```
##
```

```
clog <- mpg$manufacturer == "toyota"
```

```
mpg[clog, ]
```

Múltiplas condições lógicas podem ser construídas dentro da função `filter()`. Por exemplo, podemos ser um pouco mais específicos e selecionarmos apenas os modelos da Toyota que possuem um motor de 4 cilindradas com o comando abaixo. Repare abaixo, que ao acrescentarmos novas condições na função `filter()`, elas acabam se tornando dependentes. Ou seja, ambas as condições devem ser atendidas ao mesmo tempo em cada linha retornada pela função `filter()`.

```
mpg %>% filter(manufacturer == "toyota", cyl == 4)
```

```
## # A tibble: 18 x 11
##   manufacturer model displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 toyota      4run~   2.7  1999     4 manu~ 4     15    20 r    suv
## 2 toyota      4run~   2.7  1999     4 auto~ 4     16    20 r    suv
## 3 toyota      camry   2.2  1999     4 manu~ f     21    29 r    mids~
## 4 toyota      camry   2.2  1999     4 auto~ f     21    27 r    mids~
## 5 toyota      camry   2.4  2008     4 manu~ f     21    31 r    mids~
## 6 toyota      camry   2.4  2008     4 auto~ f     21    31 r    mids~
## 7 toyota      camr~   2.2  1999     4 auto~ f     21    27 r    comp~
## 8 toyota      camr~   2.2  1999     4 manu~ f     21    29 r    comp~
## 9 toyota      camr~   2.4  2008     4 manu~ f     21    31 r    comp~
## 10 toyota     camr~   2.4  2008     4 auto~ f     22    31 r    comp~
```

```
## 11 toyota      coro~  1.8  1999    4 auto~ f      24    30 r    comp~
## 12 toyota      coro~  1.8  1999    4 auto~ f      24    33 r    comp~
## 13 toyota      coro~  1.8  1999    4 manu~ f      26    35 r    comp~
## 14 toyota      coro~  1.8  2008    4 manu~ f      28    37 r    comp~
## 15 toyota      coro~  1.8  2008    4 auto~ f      26    35 r    comp~
## 16 toyota      toyo~  2.7  1999    4 manu~ 4      15    20 r    pick~
## 17 toyota      toyo~  2.7  1999    4 auto~ 4      16    20 r    pick~
## 18 toyota      toyo~  2.7  2008    4 manu~ 4      17    22 r    pick~
```

```
## -----
```

```
## A mesma operação por subsetting:
```

```
##
```

```
clog <- mpg$manufacturer == "toyota" & mpg$cyl == 4
```

```
mpg[clog, ]
```

Nós tradicionalmente estabelecemos relações de dependência entre condições lógicas, por meio do operador &. Mas a função `filter()` busca ser prática e, por isso, ela automaticamente realiza esse trabalho por nós. Porém, isso implica que se as suas condições forem independentes, ajustes precisam ser feitos, através do operador `|`.

Visto esse ponto, você pode estar interessado em filtrar a sua tabela, de acordo com um conjunto de valores. Por exemplo, ao invés de selecionar apenas os modelos pertencentes à Toyota, podemos selecionar um conjunto maior de marcas. Em ocasiões como essa, o operador `%in%` se torna útil, pois você está pesquisando se o valor presente em cada linha de sua tabela, pertence ou não a um dado conjunto de valores.

```
marcas <- c("volkswagen", "audi", "toyota", "honda")
```

```
mpg %>%
```

```
  filter(manufacturer %in% marcas)
```

```
## # A tibble: 88 x 11
```

```
##   manufacturer model displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto~ f     18    29 p    comp~
## 2 audi         a4      1.8  1999     4 manu~ f     21    29 p    comp~
## 3 audi         a4      2    2008     4 manu~ f     20    31 p    comp~
## 4 audi         a4      2    2008     4 auto~ f     21    30 p    comp~
## 5 audi         a4      2.8  1999     6 auto~ f     16    26 p    comp~
## 6 audi         a4      2.8  1999     6 manu~ f     18    26 p    comp~
## 7 audi         a4      3.1  2008     6 auto~ f     18    27 p    comp~
## 8 audi         a4 q~    1.8  1999     4 manu~ 4     18    26 p    comp~
## 9 audi         a4 q~    1.8  1999     4 auto~ 4     16    25 p    comp~
## 10 audi        a4 q~    2    2008     4 manu~ 4     20    28 p    comp~
```

```
## # ... with 78 more rows
```

```
## -----
## A mesma operação por subsetting:
##
marcas <- c("volkswagen", "audi", "toyota", "honda")
clog <- mpg$manufacturer %in% marcas

mpg[clog, ]
```

4.5.1 Cuidados com o operador de igualdade

Quando você estiver filtrando as linhas de sua tabela de acordo com uma condição de igualdade, é importante que você tome alguns cuidados, especialmente se valores textuais estiverem envolvidos nessa condição. O primeiro ponto a ser abordado é o uso do operador `==`, que para além de igualdade, ele busca encontrar valores **exatamente** iguais.

O “exatamente” é importante aqui, pois certos valores numéricos podem ser aparentemente idênticos aos nossos olhos, mas ainda assim, diferentes segundo a visão de `==`. Isso ocorre especialmente com valores numéricos do tipo `double`. Pois os nossos computadores utilizam precisão aritmética finita para guardar esse tipo de valor (WICKHAM; GROLEMUND, 2017, p. 47). Isso significa que os nossos computadores guardam apenas as casas decimais significantes de um valor `double`, e a perda de casas decimais que ocorre nesse processo, pode ser a fonte de alguma diferença em operações aritméticas. Por exemplo, se testarmos a igualdade entre $(\sqrt{2})^2 = 2$, o R vai nos indicar alguma diferença existente entre esses dois valores.

```
(sqrt(2)^2) == 2
```

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

Por essa razão, quando você estiver testando a igualdade entre valores do tipo `double`, é interessante que você utilize a função `near()` ao invés do operador `==`. Por padrão, a função `near()` possui uma tolerância próxima de $1,49 \times 10^{-8}$, mas você pode ajustar esse valor pelo argumento `tol` da função.

```
near(sqrt(2)^2, 2)
```

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

Para mais, você também deve estar atento ao uso do operador `==`, quando estiver testando a igualdade entre palavras, ou valores textuais. Pois uma palavra pode ser escrita de múltiplas maneiras sem que ela perca o seu sentido, e a mínima diferença presente nos caracteres utilizados pode torná-las valores completamente diferentes aos olhos do operador `==`. Logo, os valores `"Isabela"` e `"isabela"` são diferentes na visão de `==`, mesmo que na prática, esses valores muito provavelmente se referem ao mesmo indivíduo.

```
"Isabela" == "isabela"
```



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

Se você possui em sua coluna, uma variedade maior de valores textuais, que são diferentes, mas que dizem respeito ao mesmo indivíduo (por exemplo, você possui seis variedades de “Isabela”: Isabela; ISABELA; IsAbElA; Ísabela; ísabela; i\@abela), você muito provavelmente necessita de uma expressão regular. Para acessar esse mecanismo e utilizá-lo dentro da função `filter()`, você precisa de uma função que utilize essa funcionalidade para pesquisar os textos que se encaixam em sua expressão, e que retorne como resultado, um vetor de valores lógicos que indicam as linhas de sua tabela em que esses textos ocorrem. Sendo os principais indivíduos dessa categoria, a função `grepl()`, e a função `str_detect()` que pertence ao pacote `stringr`.

Por outro lado, pode ser que você não precise ir tão longe, caso as diferenças presentes em seus textos se apresentem na forma de capitalização das letras (maiúsculo ou minúsculo). Por exemplo, suponha que a sua variedade de “Isabela” fosse: Isabela; ISABELA; IsAbElA e isabela. Para tornar esses valores iguais, você precisaria apenas de um método de pesquisa que seja capaz de ignorar a capitalização das letras. Para isso, você pode utilizar a função `grepl()` que possui o argumento `ignore.case`, no qual você pode pedir a função que ignore essas diferenças na capitalização, como no exemplo abaixo.

```
set.seed(2)
df <- data.frame(
  usuario = c("Ana", "Isabela", "isabela", "Julia"),
  id = 1:4,
  valor = round(rnorm(4), 2)
)

df %>%
  filter(grepl("Isabela", usuario, ignore.case = TRUE))

##   usuario id valor
## 1 Isabela  2  0.18
## 2 isabela  3  1.59
```

4.5.2 Estabelecendo intervalos com a função `between()`

Para estabelecermos uma condição de intervalo no R, precisamos de duas condições lógicas que definam os limites deste intervalo. Em seguida, nós devemos tornar essas duas condições dependentes. Por exemplo, se desejássemos filtrar todas as linhas de `mpg` que possuem um valor na coluna `hwy` entre 18 e 24, precisaríamos do seguinte teste lógico:

```
mpg %>%
  filter(hwy >= 18, hwy <= 24)

## -----
```

```
## A mesma operação por subsetting:
##
clog <- mpg$hwy >= 18 & mpg$hwy <= 24

mpg[clog, ]
```

Porém, de uma maneira mais prática, podemos utilizar a função `between()` que consiste em um atalho para essa metodologia. A função possui três argumentos: 1) `x`, a coluna ou o vetor sobre o qual você deseja aplicar o teste de intervalo; 2) `left`, o limite “inferior” (ou “esquerdo”) do intervalo; 3) `right`, o limite “superior” (ou “direito”) do intervalo. Logo, se fôssemos traduzir o teste de intervalo anterior para a função `between()`, faríamos da seguinte maneira:

```
mpg %>%
  filter(between(hwy, 18, 24))

## # A tibble: 63 x 11
##   manufacturer model displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi          a6 q~   2.8  1999     6 auto~ 4     15    24 p   mids~
## 2 audi          a6 q~   4.2  2008     8 auto~ 4     16    23 p   mids~
## 3 chevrolet     c150~   5.3  2008     8 auto~ r     14    20 r   suv
## 4 chevrolet     c150~   5.3  2008     8 auto~ r     14    20 r   suv
## 5 chevrolet     corv~   5.7  1999     8 auto~ r     15    23 p   2sea~
## 6 chevrolet     corv~   7    2008     8 manu~ r     15    24 p   2sea~
## 7 chevrolet     k150~   5.3  2008     8 auto~ 4     14    19 r   suv
## 8 dodge         cara~   2.4  1999     4 auto~ f     18    24 r   mini~
## 9 dodge         cara~   3    1999     6 auto~ f     17    24 r   mini~
## 10 dodge        cara~   3.3  1999     6 auto~ f     16    22 r   mini~
## # ... with 53 more rows
```

4.5.3 Ataque terrorista

Vamos dar um pouco de contexto para as nossas operações. Nessa seção, vamos utilizar os dados disponíveis na tabela `transf`, que podem ser importados para o seu R através dos comandos abaixo. A tabela `transf` contém informações sobre diversas transferências bancárias realizadas por uma instituição bancária. Algumas informações presentes nessa tabela incluem: a data e o horário da transferência (`Data`); O `username` do usuário do banco responsável por realizar a transferência (`Usuario`); o país de destino da transferência (`Pais`); um código de identificação da transferência (`TransferID`); e o valor transferido (`Valor`).

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

```
library(readr)

transf <- read_csv2(paste0(github, pasta, arquivo))

transf

## # A tibble: 20,006 x 6
##   Data          Usuario Valor TransferID Pais   Descricao
##   <dtm>         <chr>    <dbl>      <dbl> <chr>    <lg1>
## 1 2018-12-06 22:19:19 Eduardo    599.   116241629 Alemanha NA
## 2 2018-12-06 22:10:34 Júlio     4611.   115586504 Alemanha NA
## 3 2018-12-06 21:59:50 Nathália  4418.   115079280 Alemanha NA
## 4 2018-12-06 21:54:13 Júlio     2740.   114972398 Alemanha NA
## 5 2018-12-06 21:41:27 Ana        1408.   116262934 Alemanha NA
## 6 2018-12-06 21:18:40 Nathália  5052.   115710402 Alemanha NA
## 7 2018-12-06 20:54:32 Eduardo    5665.   114830203 Alemanha NA
## 8 2018-12-06 20:15:46 Sandra     1474.   116323455 Alemanha NA
## 9 2018-12-06 20:04:35 Armando     8906.   115304382 Alemanha NA
## 10 2018-12-22 20:00:56 Armando   18521.   114513684 Alemanha NA
## # ... with 19,996 more rows
```

Vamos supor que no dia 24 de dezembro de 2018, tenha ocorrido um ataque terrorista na cidade de Berlim (Alemanha). Suponha também, que você faz parte do setor de *compliance* da instituição financeira responsável pelas transferências descritas na tabela `transf`. Em geral, um dos principais papéis de um setor de *compliance* é garantir que a sua instituição não esteja contribuindo com práticas ilícitas (dentre elas está o terrorismo).

Segundo o relatório da polícia, há fortes indícios de que a munição utilizada no ato, foi comprada durante os dias 20 e 23. Além disso, a polícia também destacou que levando em conta a quantidade utilizada no ataque, somente a munição empregada custou em média mais de \$15.000.

Logo, o seu papel seria se certificar de que a instituição a qual você pertence, não realizou alguma transferência que se encaixa nessas características. Pois caso tal transferência exista, vocês teriam de abrir uma investigação em conjunto com a polícia, para apurar as fontes e os destinatários dos recursos dessa transferência.

Portanto, estamos procurando por uma transferência na tabela `transf` de valor acima de \$15.000, que possua a Alemanha como país de destino, e que tenha ocorrido durante os dias 20 e 23 de dezembro de 2018. Perceba que todas essas condições, ou características da transferência devem ser atendidas ao mesmo tempo. Ou seja, essas condições lógicas são dependentes uma da outra.

Lembre-se que quando temos diversas condições lógicas dependentes, nós podemos separá-las por vírgulas na função `filter()`. Por outro lado, fora do uso da função `filter()`, nós estabelecemos uma relação de dependência entre várias condições lógicas por meio do operador `&`, e será esse o método tradicional utilizado nessa seção. Logo, quando temos diversas condições no R que devem

ser atendidas ao mesmo tempo, nós devemos conectar cada uma dessas condições pelo operador &, como no exemplo abaixo.

```
transf %>%
  filter(
    Valor > 15000 & Pais == "Alemanha" &
    between(as.Date(Data), as.Date("2018-12-20"), as.Date("2018-12-23"))
  )

## # A tibble: 132 x 6
##   Data          Usuario      Valor TransferID Pais      Descricao
##   <dtm>         <chr>      <dbl>      <dbl> <chr>      <lg1>
## 1 2018-12-22 20:00:56 Armando      18521.  114513684 Alemanha NA
## 2 2018-12-21 18:46:59 Júlio Cesar 16226.  116279014 Alemanha NA
## 3 2018-12-21 17:41:48 Nathália 17583.  115748273 Alemanha NA
## 4 2018-12-23 09:46:23 Júlio      15396.  115272184 Alemanha NA
## 5 2018-12-21 06:38:20 Júlio Cesar 17555.  114983226 Alemanha NA
## 6 2018-12-23 18:11:27 Eduardo      17219.  115904797 Alemanha NA
## 7 2018-12-22 13:09:13 Eduardo      16255.  114520578 Alemanha NA
## 8 2018-12-23 10:59:50 Júlio Cesar 15093.  115919119 Alemanha NA
## 9 2018-12-23 10:29:34 Sandra        19241.  114665132 Alemanha NA
## 10 2018-12-21 06:04:49 Júlio Cesar 18938.  116281869 Alemanha NA
## # ... with 122 more rows

## -----
## A mesma operação por subsetting:
##
clog <- transf$Valor > 15000 & transf$Pais == "Alemanha" &
  between(as.Date(transf$Data), as.Date("2018-12-20"), as.Date("2018-12-23"))

transf[clog, ]
```

No total, 132 linhas foram retornadas pela função, e você teria de conferir cada uma dessas transferências. Um baita trabalho! Porém, vamos supor que em um minuto de reflexão sobre as regras do banco, você se lembre que o remetente da transferência não é obrigado a apresentar uma prova de fundos ou um comprovante de endereço, caso a transferência possua um valor menor do que \$200. Em casos como esse, o remetente precisa apresentar apenas a identidade (que ele pode ter falsificado).

```
transf %>%
  filter(
    Valor <= 200 & Pais == "Alemanha" &
    between(as.Date(Data), as.Date("2018-12-20"), as.Date("2018-12-23"))
  )
```

```
## # A tibble: 5 x 6
##   Data                Usuario Valor TransferID Pais      Descricao
##   <dtm>              <chr>   <dbl>     <dbl> <chr>    <lgl>
## 1 2018-12-20 00:31:17 Júlio     193   115555598 Alemanha NA
## 2 2018-12-22 06:30:01 Sandra     100   116400001 Alemanha NA
## 3 2018-12-22 06:35:00 Sandra     200   116400002 Alemanha NA
## 4 2018-12-22 06:42:12 Eduardo     200   116400005 Alemanha NA
## 5 2018-12-22 06:55:54 Eduardo     150   116400009 Alemanha NA
```

Isso é interessante, pois conseguimos reduzir os nossos resultados para apenas 5 transferências. Ao conferirmos as informações da primeira transferência, os recursos estão limpos. Porém, as próximas 4 transferências levantam algumas suspeitas. Pois elas foram realizadas por clientes diferentes, mas com poucos minutos de diferença. Ao conversar com os agentes Sandra e Eduardo, que autorizaram essas transferências, você descobre que todos os diferentes clientes apresentaram transferências francesas. Será que esses clientes estavam testando as regras da instituição para com identidades desse país?

Ao procurar por todas as transferências em que identidades francesas foram apresentadas, e que foram realizadas entre os dias 20 e 23 de dezembro de 2018, e que possuíam a Alemanha como país de destino, você chega a uma estranha transferência de \$20.000 efetuada poucos minutos depois das 4 transferências que encontramos anteriormente. Durante a análise das informações dessa transferência, você percebe diversas falhas presentes na prova de fundos que sustentou a decisão de autorização dessa operação. Há uma grande possibilidade de que os chefes e agentes de sua instituição que autorizaram essa operação, estejam em maus lençóis.

```
transf %>%
  inner_join(
    identidade,
    by = "TransferID"
  ) %>%
  filter(
    Pais == "Alemanha" & Identi_Nacion == "França" &
    between(as.Date(Data), as.Date("2018-12-20"), as.Date("2018-12-23"))
  )
```

```
## # A tibble: 5 x 7
##   Data                Usuario Valor TransferID Pais      Descricao
##   <dtm>              <chr>   <dbl>     <dbl> <chr>    <lgl>
## 1 2018-12-22 06:30:01 Sandra     100   116400001 Alemanha NA
## 2 2018-12-22 06:35:00 Sandra     200   116400002 Alemanha NA
## 3 2018-12-22 06:42:12 Eduardo     200   116400005 Alemanha NA
## 4 2018-12-22 06:55:54 Eduardo     150   116400009 Alemanha NA
## 5 2018-12-22 06:59:07 Eduardo    20000  116400010 Alemanha NA
## # ... with 1 more variable: Identi_Nacion <chr>
```

4.5.4 Condições dependentes (&) ou independentes (|) ?

Na seção anterior, as condições lógicas que guiavam o nosso filtro eram dependentes entre si. Em outras palavras, as condições deveriam ser todas atendidas ao mesmo tempo. Por essa razão, nós conectamos as condições lógicas com o operador &. Porém, em algumas ocasiões as suas condições serão independentes e, por isso, devemos utilizar um outro operador para conectá-las, que é a barra vertical (|).

Por exemplo, se eu quiser encontrar todas as transferências na tabela `transf` que ocorreram no dia 13 de novembro de 2018, **ou** que possuem um valor menor que \$500, **ou** que foram autorizadas pelo agente Eduardo, eu devo construir o comando abaixo. Logo, toda linha da tabela `transf` que atenda pelo menos uma das condições que estabelecemos, é filtrada pela função `filter()`.

```
transf %>%
  filter(
    as.Date(Data) == as.Date("2018-11-13") | Valor < 500 |
    Usuario == "Eduardo"
  )
```

```
## # A tibble: 5,581 x 6
```

	Data	Usuario	Valor	TransferID	Pais	Descricao
	<dtm>	<chr>	<dbl>	<dbl>	<chr>	<lg1>
## 1	2018-12-06 22:19:19	Eduardo	599.	116241629	Alemanha	NA
## 2	2018-12-06 20:54:32	Eduardo	5665.	114830203	Alemanha	NA
## 3	2018-12-06 19:07:50	Eduardo	9561.	115917812	Alemanha	NA
## 4	2018-12-06 18:09:15	Júlio Cesar	388.	114894102	Alemanha	NA
## 5	2018-12-06 16:59:38	Eduardo	11759.	115580064	Alemanha	NA
## 6	2018-12-06 15:21:36	Eduardo	4436.	114425893	Alemanha	NA
## 7	2018-12-06 14:47:25	Ana	483.	114387526	Alemanha	NA
## 8	2018-12-06 12:59:58	Ana	207.	115615456	Alemanha	NA
## 9	2018-12-06 10:05:21	Eduardo	708.	114746955	Alemanha	NA
## 10	2018-12-06 09:50:03	Eduardo	1587.	114796170	Alemanha	NA

```
## # ... with 5,571 more rows
```

```
## -----
```

```
## A mesma operação por subsetting:
```

```
##
```

```
clog <- as.Date(transf$Data) == as.Date("2018-11-13") |
  transf$Valor < 500 | transf$Usuario == "Eduardo"
```

```
transf[clog, ]
```

4.6 Ordenando linhas com arrange()

Algumas operações que realizamos dependem diretamente da forma como as linhas de nossa tabela estão ordenadas. Em outros momentos, desejamos ordenar a nossa tabela, para rapidamente identificarmos as observações que possuem os 10 maiores valores de alguma variável ao longo da base. Ou seja, a ordenação de linhas é uma operação muito comum, e o pacote `dplyr` oferece a função `arrange()` para tal ação.

O uso da função `arrange()` é bem simples. Tudo o que você precisa fazer é listar as colunas pelas quais você deseja ordenar a base. Caso a coluna seja numérica, `arrange()` vai seguir uma ordenação numérica. Mas se essa coluna for do tipo `character`, `arrange()` vai utilizar uma ordenação alfabética para organizar os valores da coluna. Por outro lado, na hipótese dessa coluna ser do tipo `factor`, `arrange()` vai seguir a ordem presente nos “níveis” (`levels`) desse `factor`, aos quais você pode acessar pela função `levels()`.

```
mpg %>% arrange(displ)
```

```
## # A tibble: 234 x 11
##   manufacturer model  displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 honda        civic   1.6  1999     4 manu~ f     28    33 r   subc~
## 2 honda        civic   1.6  1999     4 auto~ f     24    32 r   subc~
## 3 honda        civic   1.6  1999     4 manu~ f     25    32 r   subc~
## 4 honda        civic   1.6  1999     4 manu~ f     23    29 p   subc~
## 5 honda        civic   1.6  1999     4 auto~ f     24    32 r   subc~
## 6 audi         a4      1.8  1999     4 auto~ f     18    29 p   comp~
## 7 audi         a4      1.8  1999     4 manu~ f     21    29 p   comp~
## 8 audi         a4 q~   1.8  1999     4 manu~ 4     18    26 p   comp~
## 9 audi         a4 q~   1.8  1999     4 auto~ 4     16    25 p   comp~
## 10 honda        civic   1.8  2008     4 manu~ f     26    34 r   subc~
## # ... with 224 more rows
```

Você pode recorrer a várias colunas para ordenar a sua base. Nessa situação, a função `arrange()` vai ordenar as colunas na ordem em que você as definiu na função. Ou seja, no exemplo abaixo, a função `arrange()` primeiro ordena a base de acordo com a coluna `displ`, em seguida, segundo a coluna `hwy`, e por último, a coluna `trans`.

```
mpg %>% arrange(displ, hwy, trans)
```

```
## # A tibble: 234 x 11
##   manufacturer model  displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 honda        civic   1.6  1999     4 manu~ f     23    29 p   subc~
## 2 honda        civic   1.6  1999     4 auto~ f     24    32 r   subc~
```

```
## 3 honda      civic  1.6 1999    4 auto~ f      24    32 r    subc~
## 4 honda      civic  1.6 1999    4 manu~ f      25    32 r    subc~
## 5 honda      civic  1.6 1999    4 manu~ f      28    33 r    subc~
## 6 audi       a4 q~  1.8 1999    4 auto~ 4      16    25 p    comp~
## 7 audi       a4 q~  1.8 1999    4 manu~ 4      18    26 p    comp~
## 8 audi       a4     1.8 1999    4 auto~ f      18    29 p    comp~
## 9 volkswagen pass~  1.8 1999    4 auto~ f      18    29 p    mids~
## 10 audi      a4     1.8 1999    4 manu~ f      21    29 p    comp~
## # ... with 224 more rows
```

Por padrão, a função `arrange()` utiliza uma ordenação em um sentido crescente (do menor para o maior valor; do primeiro para o último valor), qualquer que seja o tipo de dado contido na coluna que você forneceu a função. Caso você deseje utilizar uma ordenação em um sentido decrescente (do maior para o menor valor; do último para o primeiro valor) em uma dada coluna, você deve encapsular o nome dessa coluna na função `desc()`. No exemplo abaixo, `arrange()` primeiro ordena a coluna `manufacturer` em uma forma decrescente e, em seguida, ordena a coluna `hwy` de acordo com uma ordem crescente.

```
mpg %>% arrange(desc(manufacturer), hwy)
```

```
## # A tibble: 234 x 11
##   manufacturer model displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 volkswagen   jetta   2.8  1999    6 auto~ f     16    23 r    comp~
## 2 volkswagen   gti     2.8  1999    6 manu~ f     17    24 r    comp~
## 3 volkswagen   jetta   2.8  1999    6 manu~ f     17    24 r    comp~
## 4 volkswagen   gti     2    1999    4 auto~ f     19    26 r    comp~
## 5 volkswagen   jetta   2    1999    4 auto~ f     19    26 r    comp~
## 6 volkswagen   new ~    2    1999    4 auto~ f     19    26 r    subc~
## 7 volkswagen   pass~   2.8  1999    6 auto~ f     16    26 p    mids~
## 8 volkswagen   pass~   2.8  1999    6 manu~ f     18    26 p    mids~
## 9 volkswagen   pass~   3.6  2008    6 auto~ f     17    26 p    mids~
## 10 volkswagen  new ~    2.5  2008    5 manu~ f     20    28 r    subc~
## # ... with 224 more rows
```

Como estamos basicamente definindo colunas na função `arrange()`, é natural que você anseie pelos diversos métodos de seleção que aprendemos em `select()`. Por isso, em versões mais recentes do pacote `dplyr` tivemos a introdução da função `across()`, pela qual você tem novamente acesso a todos esses métodos que vimos em `select()`.

```
## Ordenar a base segundo as três primeiras colunas
mpg %>% arrange(across(1:3))
```

```
## Ordenar a base segundo o conjunto de colunas
```



```
## que possuem um nome que se inicia
## pelos caracteres "dis"
mpg %>% arrange(across(starts_with("dis")))
```

Vale destacar que a função `arrange()`, por padrão, não respeita os grupos de sua tabela e, portanto, considera toda a sua tabela no momento em que a ordenação ocorre. Ainda veremos em mais detalhes nas próximas seções, a função `group_by()`, pela qual você pode definir os grupos presentes em sua tabela. Portanto, pode ser de seu desejo que a ordenação executada por `arrange()` ocorra dentro de cada um dos grupos que você delimitou através da função `group_by()`. Para isso, você precisa configurar o argumento `.by_group` para `TRUE`.

```
mpg %>%
  group_by(manufacturer) %>%
  arrange(hwy, .by_group = TRUE)
```

```
## # A tibble: 234 x 11
## # Groups:   manufacturer [15]
##   manufacturer model displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi         a6 q~   4.2  2008     8 auto~ 4     16    23 p   mids~
## 2 audi         a6 q~   2.8  1999     6 auto~ 4     15    24 p   mids~
## 3 audi         a4 q~   1.8  1999     4 auto~ 4     16    25 p   comp~
## 4 audi         a4 q~   2.8  1999     6 auto~ 4     15    25 p   comp~
## 5 audi         a4 q~   2.8  1999     6 manu~ 4     17    25 p   comp~
## 6 audi         a4 q~   3.1  2008     6 auto~ 4     17    25 p   comp~
## 7 audi         a4 q~   3.1  2008     6 manu~ 4     15    25 p   comp~
## 8 audi         a6 q~   3.1  2008     6 auto~ 4     17    25 p   mids~
## 9 audi         a4      2.8  1999     6 auto~ f     16    26 p   comp~
## 10 audi        a4      2.8  1999     6 manu~ f     18    26 p   comp~
## # ... with 224 more rows
```

4.7 Adicionando variáveis à sua tabela com mutate()

Frequentemente, você deseja adicionar uma nova variável em sua tabela como uma função de outras variáveis já existentes em sua tabela. Para tal fim, o pacote `dplyr` disponibiliza a função `mutate()`, que oferta um mecanismo limpo e rápido para executarmos tal ação.

Como um exemplo inicial, vamos voltar a tabela `transf` que introduzimos na seção **Ataque terrorista**. A coluna `Data` retém a data e o horário em que cada operação foi registrada no sistema do banco. Entretanto, o horário pode se tornar irrelevante para certos passos e, por essa razão, seria interessante que possuíssemos uma coluna na tabela `transf`, contendo apenas a data de cada transferência. Com esse objetivo em mente, somos capazes de extrair a data da coluna `Data` através da função `as.Date()`, e empregar a função `mutate()` para armazenarmos o resultado desse procedimento em uma nova coluna chamada `Sem_hora`, como mostrado abaixo.

```

transf %>%
  select(-Pais, -Descricao) %>%
  mutate(
    Sem_hora = as.Date(Data)
  )

## # A tibble: 20,006 x 5
##   Data          Usuario Valor TransferID Sem_hora
##   <dtm>         <chr>   <dbl>      <dbl> <date>
## 1 2018-12-06 22:19:19 Eduardo    599.    116241629 2018-12-06
## 2 2018-12-06 22:10:34 Júlio     4611.    115586504 2018-12-06
## 3 2018-12-06 21:59:50 Nathália  4418.    115079280 2018-12-06
## 4 2018-12-06 21:54:13 Júlio     2740.    114972398 2018-12-06
## 5 2018-12-06 21:41:27 Ana       1408.    116262934 2018-12-06
## 6 2018-12-06 21:18:40 Nathália  5052.    115710402 2018-12-06
## 7 2018-12-06 20:54:32 Eduardo   5665.    114830203 2018-12-06
## 8 2018-12-06 20:15:46 Sandra    1474.    116323455 2018-12-06
## 9 2018-12-06 20:04:35 Armando   8906.    115304382 2018-12-06
## 10 2018-12-22 20:00:56 Armando  18521.    114513684 2018-12-22
## # ... with 19,996 more rows

```

Portanto, sempre que você recorrer à função `mutate()`, você deve compor essa estrutura de `<nome_coluna> = <expressao>` em cada coluna adicionada. Ou seja, como flexibilidade e eficiência são valores que as funções do pacote `dplyr` carregam, você tem a capacidade de criar múltiplas colunas em um mesmo `mutate()`. Porém, como um aviso, é ideal que você não crie mais de 7 colunas ao mesmo tempo. Na hipótese dessa recomendação ser ignorada, há uma probabilidade significativa de você enfrentar problemas de memória e mensagens de erro bastante nebulosas.

```

## Estrutura básica de um mutate():
<sua_tabela> %>%
  mutate(
    nome_coluna1 = expressao1,
    nome_coluna2 = expressao2,
    nome_coluna3 = expressao3,
    ...
  )

```

Um outro ponto muito importante, é que em um mesmo `mutate()`, você também pode empregar uma nova coluna que você acaba de criar, no cálculo de uma outra coluna a ser produzida. Por exemplo, eu posso guardar o desvio de Valor em relação à sua média, na coluna `Desvio`, e logo em seguida, utilizar os valores dessa coluna para produzir a coluna `Valor_norm`, como exposto abaixo.

```

transf %>%
  select(-Pais, -Descricao) %>%

```

```
mutate(
  Desvio = Valor - mean(Valor),
  Valor_norm = Desvio / sd(Valor)
)

## # A tibble: 20,006 x 6
##   Data          Usuario Valor TransferID Desvio Valor_norm
##   <dtm>         <chr>   <dbl>     <dbl>  <dbl>    <dbl>
## 1 2018-12-06 22:19:19 Eduardo   599.  116241629 -2920.    -0.772
## 2 2018-12-06 22:10:34 Júlio    4611.  115586504  1093.     0.289
## 3 2018-12-06 21:59:50 Nathália 4418.  115079280   900.     0.238
## 4 2018-12-06 21:54:13 Júlio    2740.  114972398  -778.    -0.206
## 5 2018-12-06 21:41:27 Ana      1408.  116262934 -2110.    -0.558
## 6 2018-12-06 21:18:40 Nathália 5052.  115710402  1534.     0.405
## 7 2018-12-06 20:54:32 Eduardo   5665.  114830203  2147.     0.568
## 8 2018-12-06 20:15:46 Sandra    1474.  116323455 -2044.    -0.540
## 9 2018-12-06 20:04:35 Armando   8906.  115304382  5387.     1.42
## 10 2018-12-22 20:00:56 Armando  18521.  114513684 15003.     3.97
## # ... with 19,996 more rows
```

Com isso, a parte fundamental de um `mutate()` é construímos a expressão que produzirá os valores a serem alocados na nova coluna que estamos criando. Logo abaixo, consta uma lista de várias funções que você pode utilizar para formar a expressão que você deseja. Ademais, essa é uma lista parcial, logo, há diversas outras funções que você pode utilizar para calcular os valores dos quais você necessita.

1. **Somatórios:** soma total de uma coluna - `sum()`; somatório por linha, ao longo de algumas colunas - operador `+`; somatório por linha, ao longo de várias colunas - `rowSums()`.
2. **Operações cumulativas:** somatório acumulado de uma coluna - `cumsum()`; média acumulada de uma coluna - `cummean()`; mínimo acumulado de uma coluna - `cummin()`; máximo acumulado de uma coluna - `cummax()`.
3. **Medidas de posição:** média de uma coluna - `mean()`; mediana de uma coluna - `median()`; média por linha, ao longo de várias colunas - `rowMeans()`; média móvel - `roll_mean()`¹.
4. **Medidas de dispersão:** desvio padrão de uma coluna - `sd()`; variância de uma coluna - `var()`; intervalo interquartil - `IQR()`; desvio absoluto da mediana - `mad()`.
5. **Operadores aritméticos:** soma (`+`); subtração (`-`); divisão (`/`); multiplicação (`*`); potência, ou elevar um número a `x` (`^`); restante da divisão (`%`); apenas o número inteiro resultante da divisão (`%/%`); logaritmo - `log()`.
6. **Operadores lógicos:** aplique um teste lógico em cada linha, e preencha essa linha com `x` caso o teste resulte em `TRUE`, ou preencha com `y` caso o teste resulte em `FALSE` - `if_else()`;

¹Essa função pertence ao pacote `RcppRoll` e, portanto, para ter acesso à função você deve possuir esse pacote instalado em sua máquina, e chamar por ele em sua sessão.

quando você quer aplicar uma operação parecida com `if_else()`, mas que há vários casos possíveis, um exemplo típico seria criar uma coluna de faixas etárias - `case_when()`; você também pode utilizar normalmente todos os operadores que vimos na seção de `filter()`, para criar um teste lógico sobre cada linha - `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`, `&`, `|`.

7. **Funções para discretizar variáveis contínuas:** calcula intervalos de forma a encaixar o mesmo número de observações em cada intervalo (comumente chamados de quantis) - `cut_number()`; calcula intervalos com o mesmo alcance - `cut_interval()`; calcula intervalos de largura definida no argumento `width` - `cut_width()`.
8. **Funções de defasagem e liderança:** quando você precisa em algum cálculo naquela linha, utilizar o valor da linha anterior - `lag()`; ou ao invés do valor da linha anterior, você precisa do valor da linha posterior - `lead()`.

Porém, é necessário ter cautela. Como a função `mutate()` busca trabalhar com `data.frame`'s, é de suma importância, que você esteja sempre consciente das propriedades que essa estrutura carrega. Em especial, a propriedade de que as suas colunas devem possuir o mesmo número de elementos. Portanto, se o seu `data.frame` possui exatamente 10 mil linhas, você precisa se certificar de que cada expressão utilizada na função `mutate()`, vai gerar 10 mil elementos como resultado.

Na hipótese de que alguma dessas expressões produzam, por exemplo, 9.999 elementos, um erro será acionado, pois esses 9.999 mil elementos não podem ser guardados em um `data.frame` que possui 10 mil linhas. Logo, a função `mutate()` lhe provê flexibilidade e eficiência, mas ela não é capaz de quebrar regras fundamentais da linguagem R.

Um exemplo prático disso é encontrado quando tentamos calcular uma média móvel de alguma série temporal, ou de algum valor diário utilizando a função `mutate()`, como no exemplo abaixo. O erro ocorre devido a própria natureza do cálculo de uma média móvel, que gera uma “perda” de observações, e como consequência, um número menor de observações é gerado dentro do resultado. Perceba abaixo, que ao aplicarmos uma janela de cálculo de 5 observações, a função `roll_mean()` foi capaz de produzir 996 valores, consequentemente, perdemos 4 observações no processo.

```
set.seed(1)
df <- tibble(
  dia = 1:1000,
  valor = rnorm(1000)
)

library(RcppRoll)

df %>%
  mutate(
    media_movel = roll_mean(df$valor, n = 5)
  )
```

```
Erro: Problem with `mutate()` input `media_movel`.
x Input `media_movel` can't be recycled to size 1000.
```

```
i Input `media_movel` is `roll_mean(df$valor, n = 5)`.
i Input `media_movel` must be size 1000 or 1, not 996.
Run `rlang::last_error()` to see where the error occurred.
```

Compreendendo os potenciais problemas fabricados por essa característica do cálculo de uma média móvel, a função `roll_mean()` oferece o argumento `fill`, no qual podemos pedir à função que complete as observações restantes com zeros, como no exemplo abaixo. Dessa forma, a função volta a produzir 1000 observações em seu resultado e, conseqüentemente, nenhum erro é acionado.

```
df %>%
  mutate(
    media_movel = roll_mean(valor, n = 5, fill = 0, align = "right")
  )

## # A tibble: 1,000 x 3
##       dia  valor media_movel
##   <int> <dbl>      <dbl>
## 1     1 -0.626         0
## 2     2  0.184         0
## 3     3 -0.836         0
## 4     4  1.60         0
## 5     5  0.330        0.129
## 6     6 -0.820        0.0905
## 7     7  0.487        0.151
## 8     8  0.738        0.466
## 9     9  0.576        0.262
## 10    10 -0.305        0.135
## # ... with 990 more rows
```

Desse modo, estamos discutindo as possibilidades existentes com a hipótese de sua expressão fornecida à `mutate()`, produzir múltiplos valores. Todavia, diversas funções extremamente úteis, e que utilizamos com bastante frequência nessas expressões, resultam apenas em um único valor. Grandes exemplos são as funções `mean()` e `sum()`, que calculam a média e a soma de uma coluna, respectivamente.

Em todas as ocasiões em que a sua expressão na função `mutate()` gerar um único valor, qualquer que ele seja, a função `mutate()` irá automaticamente replicar esse mesmo valor ao longo de toda a coluna que você acaba de criar. Vemos uma demonstração disso, ao criarmos abaixo, as colunas `soma`, `prop` e `um_numero`. Com essa ideia em mente, se temos diversos valores numéricos em uma dada coluna, nós podemos eficientemente calcular uma proporção desses valores em relação ao total de sua coluna, com o uso da função `sum()`, como no exemplo abaixo. Da mesma forma, nós podemos rapidamente normalizar uma coluna numérica segundo a fórmula de uma estatística Z, por meio das funções `sd()` e `mean()`.

```
df <- tibble(
  id = 1:5,
```

```

x = c(2.5, 1.5, 3.2, 5.1, 2.2),
y = c(1, 2, 3, 4, 5)
)

df <- df %>%
  mutate(
    soma = sum(x),
    prop = y * 100 / sum(y),
    um_numero = 25,
    norm = (x - mean(x)) / sd(x)
  )

df

## # A tibble: 5 x 7
##       id     x     y soma prop um_numero  norm
##   <int> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
## 1     1  2.5     1  14.5  6.67     25 -0.291
## 2     2  1.5     2  14.5 13.3     25 -1.02
## 3     3  3.2     3  14.5  20      25  0.219
## 4     4  5.1     4  14.5 26.7     25  1.60
## 5     5  2.2     5  14.5 33.3     25 -0.510

```

4.8 Agrupando dados e gerando estatísticas sumárias com group_by() e summarise()

Em diversas áreas, é muito comum que tenhamos variáveis qualitativas em nossa base de dados. Variáveis desse tipo, usualmente definem grupos ou estratos de uma amostra, população ou medida, como faixas etárias ou faixas de valor salarial. Se você está analisando, por exemplo, dados epidemiológicos, você em geral deseja examinar se uma dada doença está ocorrendo com maior ou menor intensidade em um determinado grupo de sua população.

Ou seja, será que fatores como a raça, a idade, o gênero, a orientação sexual ou a localidade de um indivíduo são capazes de afetar as suas chances de ser infectado por essa doença? De outra maneira, será que essas variáveis qualitativas são capazes de gerar, por exemplo, diferenças no salário deste indivíduo? Da mesma forma, quando analisamos a performance de determinadas firmas, desejamos saber se a localidade, o setor, o tamanho, o investimento e a receita total, além do número de funcionários dessa firma são capazes de prover alguma vantagem em relação aos seus concorrentes.

Para esse tipo de estudo, o pacote dplyr nos oferece a função group_by() que fundamentalmente altera o comportamento de funções como mutate() e summarise(), e nos permite calcular estatísticas e aplicarmos operações dentro de cada grupo presente em nossos dados. Como um exemplo inicial, vamos utilizar a tabela minas_pop, que contém dados de população e PIB (Produto Interno Bruto) dos 853 municípios do estado de Minas Gerais.

```

github <- "https://raw.githubusercontent.com/pedropark99/"
pasta <- "Curso-R/master/Dados/"
arquivo <- "populacao.csv"

minas_pop <- read_csv2(paste0(github, pasta, arquivo))

## # A tibble: 853 x 7
##   IBGE2    IBGE Munic      Populacao  Ano      PIB Intermediaria
##   <dbl>  <dbl> <chr>          <dbl> <dbl>    <dbl> <chr>
## 1    10 310010 Abadia dos Dourados    6972  2017 33389769 Uberlândia
## 2    20 310020 Abaeté            23223 2017 96201158 Divinópolis
## 3    30 310030 Abre Campo        13465 2017 29149429 Juiz de Fora
## 4    40 310040 Acaiaca           3994 2017  2521892 Juiz de Fora
## 5    50 310050 Açucena           9575 2017 15250077 Ipatinga
## 6    60 310060 Água Boa          13600 2017 29988906 Teófilo Otoni
## 7    70 310070 Água Comprida       2005 2017 74771408 Uberaba
## 8    80 310080 Aguanil           4448 2017 15444038 Varginha
## 9    90 310090 Águas Formosas    19166 2017 11236696 Teófilo Otoni
## 10  100 310100 Águas Vermelhas    13477 2017 48088397 Teófilo Otoni
## # ... with 843 more rows

```

Como demonstramos na seção anterior, a função `sum()` serve para calcularmos o total de uma coluna inteira. Logo, se aplicássemos a função `sum()` sobre a coluna `Populacao`, teríamos a população total do estado de Minas Gerais. Porém, e se desejássemos calcular a população total de cada uma das regiões intermediárias (presentes na coluna `Intermediaria`) que compõe o estado de Minas Gerais?

Para isso, nós podemos utilizar a função `group_by()` para determinar onde em nossa tabela se encontram os grupos de nossos dados. No nosso caso, esses grupos estão na coluna `Intermediaria`. Dessa forma, após utilizarmos o `group_by()`, percebeba abaixo que os totais calculados pela função `sum()`, e que estão apresentados na coluna `Pop_total`, variam ao longo da tabela de acordo com o valor presente na coluna `Intermediaria`. Logo, temos agora a população total de cada região intermediária na coluna `Pop_total`. Da mesma maneira, ao invés de possuímos uma proporção baseada na população do estado, as proporções de cada município expostas na coluna `Prop_pop_mun` possuem como denominador, a população total da região intermediária a qual o município pertence.

```

minas_pop %>%
  select(-Ano, -PIB) %>%
  group_by(Intermediaria) %>%
  mutate(
    Pop_total = sum(Populacao),
    Prop_pop_mun = Populacao * 100 / Pop_total
  )

## # A tibble: 853 x 7
## # Groups:   Intermediaria [13]

```

```
##      IBGE2      IBGE Munic      Populacao Intermediaria Pop_total Prop_pop_mun
##      <dbl>      <dbl> <chr>      <dbl> <chr>      <dbl>      <dbl>
## 1      10 310010 Abadia dos ~      6972 Uberlândia      1161513      0.600
## 2      20 310020 Abaeté      23223 Divinópolis      1300658      1.79
## 3      30 310030 Abre Campo      13465 Juíz de Fora      2334530      0.577
## 4      40 310040 Acaiaca      3994 Juíz de Fora      2334530      0.171
## 5      50 310050 Açucena      9575 Ipatinga      1022384      0.937
## 6      60 310060 Água Boa      13600 Teófilo Otoni      1222050      1.11
## 7      70 310070 Água Compri~      2005 Uberaba      800412      0.250
## 8      80 310080 Aguanil      4448 Varginha      1634643      0.272
## 9      90 310090 Águas Formo~      19166 Teófilo Otoni      1222050      1.57
## 10     100 310100 Águas Verme~      13477 Teófilo Otoni      1222050      1.10
## # ... with 843 more rows
```

Para verificarmos se os grupos em uma dada tabela estão definidos, podemos observar se a descrição Groups se encontra logo abaixo às dimensões da tabela (tibble: 853 x 7). Essa descrição Groups, acaba nos informando a coluna (ou o conjunto de colunas) envolvidas nessa definição, além do número de grupos que estão contidos em nossa tabela. Logo, pelo resultado do exemplo acima, temos 13 grupos, ou 13 regiões intermediárias diferentes presentes na coluna Intermediaria.

Como um outro exemplo, dessa vez, em um contexto mais atual, podemos utilizar os dados de COVID-19 presentes na tabela abaixo, denominada covid. Nessa tabela, temos o acumulado do número de casos confirmados do vírus em cada estado brasileiro, durante o período de 25 de Fevereiro a 27 de Julho de 2020.

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

covid <- read_csv2(paste0(github, pasta, arquivo))
```

```
## # A tibble: 3,625 x 4
##   data      estado casos mortes
##   <date>      <chr> <dbl> <dbl>
## 1 2020-03-17 AC          3      0
## 2 2020-03-18 AC          3      0
## 3 2020-03-19 AC          4      0
## 4 2020-03-20 AC          7      0
## 5 2020-03-21 AC         11      0
## 6 2020-03-22 AC         11      0
## 7 2020-03-23 AC         17      0
## 8 2020-03-24 AC         21      0
## 9 2020-03-25 AC         23      0
## 10 2020-03-26 AC         23      0
## # ... with 3,615 more rows
```


Durante o ano de 2020, a Fundação João Pinheiro (FJP) tem oferecido parte de seu corpo técnico para a Secretaria Estadual de Saúde, com o objetivo de dar suporte técnico à instituição no monitoramento das estatísticas de contaminação e impacto do vírus no estado de Minas Gerais.

Portanto, uma atividade muito comum com os dados da COVID-19, seria calcularmos a variação diária no número de casos acumulados. Tal cálculo pode ser atingido, através dos valores acumulados na coluna casos, ao subtrairmos do valor da linha corrente, o valor da linha anterior nessa mesma coluna. Para incluirmos o valor da linha anterior em nosso cálculo, podemos usar a função `lag()`, como no código abaixo:

```
covid %>%
  mutate(
    casos_var = casos - lag(casos),
    mortes_var = mortes - lag(mortes)
  )
```

Porém, temos um problema nessa operação, que emerge do fato de que não delimitamos os grupos da tabela. Por essa razão, a função `mutate()` vai aplicar a expressão `casos - lag(casos)` sobre toda a tabela de uma vez só. O correto, seria que nós aplicássemos essa operação separadamente sobre os dados de cada estado.

Dito de outra forma, ao não dizermos que cada estado deveria ser tratado de forma separada dos demais, estamos invadindo os limites de cada estado com o cálculo pertencente a outros estados. Em outras palavras, o problema que emerge do código anterior, em que não definimos os grupos, se encontra nas linhas que definem os limites entre cada estado, ou as linhas que marcam a transição entre os dados do estado A para os dados do estado B. Logo, caso não definirmos esses grupos, estaremos utilizando no cálculo da variação presente na primeira linha referente ao estado de São Paulo, o número acumulado de casos localizado na última linha pertencente ao estado que vem antes de São Paulo na base (o estado de Sergipe).

Por isso, ao utilizarmos a função `group_by()` sobre a tabela `covid`, faremos com que a função `mutate()` esteja consciente dos limites entre os dados de cada estado, e que portanto, respeite esses limites durante o cálculo dessa variação.

```
covid_novo <- covid %>%
  group_by(estado) %>%
  mutate(
    casos_var = casos - lag(casos),
    mortes_var = mortes - lag(mortes)
  )

covid_novo
```

```
## # A tibble: 3,625 x 6
## # Groups:   estado [27]
```

```
##      data      estado casos mortes casos_var mortes_var
##      <date>      <chr>  <dbl>  <dbl>      <dbl>      <dbl>
##  1 2020-03-17 AC          3      0         NA         NA
##  2 2020-03-18 AC          3      0          0          0
##  3 2020-03-19 AC          4      0          1          0
##  4 2020-03-20 AC          7      0          3          0
##  5 2020-03-21 AC         11      0          4          0
##  6 2020-03-22 AC         11      0          0          0
##  7 2020-03-23 AC         17      0          6          0
##  8 2020-03-24 AC         21      0          4          0
##  9 2020-03-25 AC         23      0          2          0
## 10 2020-03-26 AC         23      0          0          0
## # ... with 3,615 more rows
```

Agora que vimos a função `group_by()`, podemos prosseguir para a função `summarise()`, que busca sumarizar, sintetizar ou reduzir múltiplos valores de seu `data.frame` em poucas linhas. Logo, se eu aplicar a função `summarise()` sobre a tabela `minas_pop`, um novo `data.frame` será gerado, e ele irá conter provavelmente uma única linha. O seu trabalho é definir os valores que vão ocupar esse espaço.

Por isso, dentro da função `summarise()`, devemos fornecer expressões, exatamente da mesma forma que fornecemos em `mutate()`. Essas expressões vão ser responsáveis por calcular os valores que vão preencher as linhas presentes no novo `data.frame` criado. Se as expressões delineadas por você gerarem um único valor ou uma única estatística sumária, o novo `data.frame` resultante de `summarise()` vai possuir uma única linha, e uma coluna para cada expressão definida. Como exemplo, podemos calcular o somatório total e a média da coluna `Populacao` da seguinte forma:

```
minas_pop %>%
  summarise(
    total_pop = sum(Populacao),
    media_pop = mean(Populacao)
  )

## # A tibble: 1 x 2
##   total_pop media_pop
##       <dbl>    <dbl>
## 1  21040662  24667.
```

Por outro lado, caso a sua expressão produza n valores como resultado, o novo `data.frame` fabricado por `summarise()` vai possuir n linhas para alocar esses valores. Em outras palavras, o número de linhas presente no `data.frame` resultante, nesse caso, depende diretamente da quantidade de valores produzidos por sua expressão. Como um exemplo disso, podemos utilizar a função `quantile()` para extrairmos os limites do intervalo interquartil (percentis de número 25 e 75) da coluna `Populacao`.

```

minas_pop %>%
  summarise(iqr = quantile(Populacao, probs = c(0.25, 0.75)))

## # A tibble: 2 x 1
##   iqr
##   <dbl>
## 1  4844
## 2 17739

```

Apesar dessas características, a função `summarise()` é normalmente utilizada em conjunto com a função `group_by()`. Pois ao definirmos os grupos de nossa tabela, a função `summarise()` passa a produzir uma linha para cada grupo presente em nossa tabela. Logo, o cálculo da população total e da população média anterior, que produzia uma única linha, passa a gerar 13 valores diferentes e, portanto, 13 linhas diferentes ao agruparmos os dados de acordo com a coluna Intermediária. Podemos ainda aplicar a função `n()`, com o objetivo de descobrirmos quantas linhas, ou quantos municípios representam cada região intermediária do estado.

```

minas_pop %>%
  group_by(Intermediaria) %>%
  summarise(
    total_pop = sum(Populacao),
    media_pop = mean(Populacao),
    numero_municipios = n()
  )

## # A tibble: 13 x 4
##   Intermediaria      total_pop media_pop numero_municipios
##   <chr>           <dbl>     <dbl>           <int>
## 1 Barbacena       772694    15769.           49
## 2 Belo Horizonte  6237890   84296.           74
## 3 Divinópolis     1300658   21322.           61
## 4 Governador Valadares 771775   13306.           58
## 5 Ipatinga        1022384   23236            44
## 6 Juiz de Fora    2334530   15990.          146
## 7 Montes Claros   1673263   19457.           86
## 8 Patos de Minas   819435    24101.           34
## 9 Pouso Alegre    1289415   16118.           80
## 10 Teófilo Otoni   1222050   14210.           86
## 11 Uberaba         800412    27600.           29
## 12 Uberlândia     1161513   48396.           24
## 13 Varginha        1634643   19935.           82

```

Neste momento, vale a pena comentar também, a função `count()`, que se traduz como um atalho para a junção das funções `group_by()`, `summarise()` e `n()`. Logo, ao invés de construirmos

toda a estrutura de `group_by()` e `summarise()`, nós poderíamos rapidamente contabilizar o número de municípios em cada região intermediária, através da função `count()`, como no exemplo abaixo. Lembrando que cada coluna fornecida à `count()`, será repassada a `group_by()` e, portanto, será responsável por definir os grupos nos quais a contagem será aplicada. Logo, se definíssemos a função como `count(minas_pop, Intermediaria, Ano)`, estaríamos calculando o número de municípios existentes em cada região intermediária, dentro de um dado ano descrito em nossa tabela.

```
minas_pop %>% count(Intermediaria)
```

```
## # A tibble: 13 x 2
##   Intermediaria      n
##   <chr>          <int>
## 1 Barbacena        49
## 2 Belo Horizonte   74
## 3 Divinópolis      61
## 4 Governador Valadares 58
## 5 Ipatinga         44
## 6 Juíz de Fora     146
## 7 Montes Claros    86
## 8 Patos de Minas   34
## 9 Pouso Alegre     80
## 10 Teófilo Otoni    86
## 11 Uberaba         29
## 12 Uberlândia      24
## 13 Varginha        82
```

```
## -----
## Sem o uso de pipe ( %>% ) teríamos:
count(minas_pop, Intermediaria)
```

Para além desses pontos, vale destacar que certos momentos em que você necessita de várias colunas para identificar um único grupo de sua tabela, não são incomuns. Por isso, você pode incluir mais de uma coluna dentro da função `group_by()`. Por exemplo, suponha que você possua na tabela `covid`, uma coluna que apresenta o mês ao qual cada linha se encontra. Suponha ainda, que você deseja calcular a média mensal de novos casos diários em cada estado. Para realizar essa ação, você precisa aplicar o cálculo da média não apenas dentro de cada estado, mas também, dentro de cada mês disponível na base. Logo, precisamos fornecer tanto a coluna estado quanto a coluna mes à função `group_by()`, como no exemplo abaixo.

```
covid_novo %>%
  ungroup() %>%
  mutate(mes = as.integer(format(data, "%m"))) %>%
  group_by(estado, mes) %>%
  summarise(
```

```

media_novos_casos = mean(casos_var, na.rm = T)
)

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

## # A tibble: 136 x 3
## # Groups:   estado [27]
##   estado   mes media_novos_casos
##   <chr> <int>         <dbl>
## 1 AC      3          2.79
## 2 AC      4         12.1
## 3 AC      5        188.
## 4 AC      6        234.
## 5 AC      7        205.
## 6 AL      3          0.85
## 7 AL      4         34.2
## 8 AL      5        298.
## 9 AL      6        856.
## 10 AL     7        750.
## # ... with 126 more rows

```

Perceba também acima, que utilizamos a função `ungroup()` sobre a tabela `covid_novo`, antes de aplicarmos a função `group_by()`. O motivo para tal operação, está no fato de que a tabela `covid_novo` já se encontrava agrupada desde o momento em que ela foi criada. Por isso, ao aplicarmos novamente a função `group_by()` com o uso das colunas `estado` e `mes`, para redefinirmos os grupos da tabela, precisamos remover a definição anterior desses grupos. Sendo esta a única ação executada pela função `ungroup()`.

Portanto, a partir do momento em que você “terminou” de utilizar as operações “por grupo” em sua tabela, e deseja ignorar novamente esses grupos em suas próximas etapas, você deve retirar a definição dos grupos de sua tabela, por meio da função `ungroup()`.

4.9 A função `across()` como a grande novidade

A função `across()` foi uma das grandes novidades introduzidas em uma versão recente do pacote `dplyr` (WICKHAM, 2020). Essa função lhe permite aplicar os métodos de seleção que vimos em `select()`, dentro dos demais verbos expostos (funções `mutate()`, `summarise()` e `arrange()`). Para mais, o principal objetivo dessa função está em prover uma maneira muito mais prática de empregarmos uma mesma operação ao longo de várias colunas.

Por exemplo, suponha que você desejasse logaritmizar todas as colunas numéricas da tabela `mpg`. Temos então, que aplicar a mesma operação sobre 5 colunas diferentes, mais especificamente, as colunas `displ`, `year`, `cyl`, `cty` e `hwy`. Com o que vimos até o momento, você provavelmente faria tal ação da seguinte forma:

```
mpg %>%
  mutate(
    displ = log(displ),
    year = log(year),
    cyl = log(cyl),
    cty = log(cty),
    hwy = log(hwy)
  )
```

Porém, além de tedioso, a repetição envolvida nesse tipo de solução, incorre em uma grande chance de erro de nossa parte. Pois os nossos olhos tendem a prestar atenção no que é diferente dos demais, no que se destaca do ambiente, e não sobre blocos e blocos de comandos que são basicamente idênticos.

Com isso, a função `across()` provê um excelente mecanismo para automatizarmos essa aplicação. Nessa função, temos dois argumentos principais a serem preenchidos: 1) `cols`, que representa o conjunto de colunas onde a ação desejada será aplicada; 2) `fns`, a função ou a expressão que será empregada em cada coluna (neste argumento, você pode fornecer apenas o nome da função). Com isso, poderíamos reescrever a operação anterior como:

```
## Aplicar log() na terceira, quarta,
## quinta, oitava e nona coluna da tabela mpg:
mpg %>%
  mutate(across(.cols = c(3:5, 8:9), .fns = log))
```

```
## # A tibble: 234 x 11
##   manufacturer model displ  year  cyl trans drv      cty  hwy fl   class
##   <chr>          <chr> <dbl> <dbl> <dbl> <chr> <chr> <dbl> <dbl> <chr> <chr>
## 1 audi          a4    0.588 7.60  1.39 auto~ f      2.89  3.37 p    comp~
## 2 audi          a4    0.588 7.60  1.39 manu~ f      3.04  3.37 p    comp~
## 3 audi          a4    0.693 7.60  1.39 manu~ f      3.00  3.43 p    comp~
## 4 audi          a4    0.693 7.60  1.39 auto~ f      3.04  3.40 p    comp~
## 5 audi          a4    1.03  7.60  1.79 auto~ f      2.77  3.26 p    comp~
## 6 audi          a4    1.03  7.60  1.79 manu~ f      2.89  3.26 p    comp~
## 7 audi          a4    1.13  7.60  1.79 auto~ f      2.89  3.30 p    comp~
## 8 audi          a4 q~ 0.588 7.60  1.39 manu~ 4      2.89  3.26 p    comp~
## 9 audi          a4 q~ 0.588 7.60  1.39 auto~ 4      2.77  3.22 p    comp~
## 10 audi         a4 q~ 0.693 7.60  1.39 manu~ 4      3.00  3.33 p    comp~
## # ... with 224 more rows
```

Portanto, em `across()` você é capaz de aplicar qualquer um dos 5 métodos que vimos em `select()`. Como um outro exemplo, podemos aplicar a função `log()` sobre qualquer coluna que se inicie pela letra “h”, com o comando abaixo:

```
mpg %>%
  mutate(across(.cols = starts_with("h"), .fns = log))
```

```
## # A tibble: 234 x 11
##   manufacturer model displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <dbl> <chr> <chr>
## 1 audi          a4      1.8  1999    4 auto~ f     18  3.37 p   comp~
## 2 audi          a4      1.8  1999    4 manu~ f     21  3.37 p   comp~
## 3 audi          a4      2    2008    4 manu~ f     20  3.43 p   comp~
## 4 audi          a4      2    2008    4 auto~ f     21  3.40 p   comp~
## 5 audi          a4      2.8  1999    6 auto~ f     16  3.26 p   comp~
## 6 audi          a4      2.8  1999    6 manu~ f     18  3.26 p   comp~
## 7 audi          a4      3.1  2008    6 auto~ f     18  3.30 p   comp~
## 8 audi          a4 q~    1.8  1999    4 manu~ 4     18  3.26 p   comp~
## 9 audi          a4 q~    1.8  1999    4 auto~ 4     16  3.22 p   comp~
## 10 audi         a4 q~    2    2008    4 manu~ 4     20  3.33 p   comp~
## # ... with 224 more rows
```

Por outro lado, caso você necessite aplicar várias funções em cada coluna, é melhor que você crie uma nova função (a partir da palavra-chave `function`) dentro de `across()`, contendo as operações que você deseja aplicar. Pois dessa maneira, você possui um melhor controle sobre em que partes do cálculo, os valores de cada coluna serão posicionados.

Por exemplo, podemos normalizar todas as colunas numéricas da tabela `mpg`, por uma estatística Z. Perceba abaixo, que nesse caso, precisamos utilizar o valor da coluna em 3 ocasiões: duas vezes no numerador, para calcularmos o desvio de cada valor da coluna em relação a sua média; e uma vez no denominador, para calcularmos o desvio padrão. Repare também, que ao menos quatro funções são utilizadas dentro desse cálculo: as funções `mean()` e `sd()`, além dos operadores de subtração (-) e de divisão (/).

```
mpg %>%
  mutate(across(
    .cols = where(is.numeric),
    .fns = function(x) x - mean(x) / sd(x)
  ))

## # A tibble: 234 x 11
##   manufacturer model displ  year  cyl trans  drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <dbl> <dbl> <chr> <chr> <dbl> <dbl> <chr>
## 1 audi          a4    -0.887 1555.  0.346 auto(l~ f     14.0  25.1 p
## 2 audi          a4    -0.887 1555.  0.346 manual~ f     17.0  25.1 p
## 3 audi          a4    -0.687 1564.  0.346 manual~ f     16.0  27.1 p
## 4 audi          a4    -0.687 1564.  0.346 auto(a~ f     17.0  26.1 p
## 5 audi          a4     0.113 1555.  2.35  auto(l~ f     12.0  22.1 p
## 6 audi          a4     0.113 1555.  2.35  manual~ f     14.0  22.1 p
## 7 audi          a4     0.413 1564.  2.35  auto(a~ f     14.0  23.1 p
## 8 audi          a4 quat~ -0.887 1555.  0.346 manual~ 4     14.0  22.1 p
## 9 audi          a4 quat~ -0.887 1555.  0.346 auto(l~ 4     12.0  21.1 p
```

```
## 10 audi          a4 quat~ -0.687 1564. 0.346 manual~ 4      16.0  24.1 p
## # ... with 224 more rows, and 1 more variable: class <chr>
```

Com isso, a função `summarise()` também se torna um local extremamente útil para o emprego da função `across()`. Pois através de `across()`, nós podemos rapidamente aplicar uma função sobre cada coluna que desejamos sintetizar com `summarise()`. Por exemplo, somos capazes de extrair o valor total de todas as colunas numéricas da tabela `mpg`, por meio dos seguintes comandos:

```
mpg %>%
  group_by(cyl) %>%
  summarise(across(
    .cols = where(is.numeric),
    .fns = sum
  ))

## # A tibble: 4 x 5
##   cyl displ  year  cty  hwy
##   <int> <dbl> <int> <int> <int>
## 1     4  174. 162243 1702 2333
## 2     5   10   8032   82  115
## 3     6  269. 158227 1281 1803
## 4     8  359. 140317  880 1234
```

4.10 Removendo duplicatas com `distinct()`

As vezes, os nossos dados chegam com algum erro de registro, e usualmente, esse erro se manifesta na forma de registros duplicados. Nessa seção, veremos o uso da função `distinct()` como um mecanismo útil para eliminarmos observações duplicadas em sua tabela. Como um exemplo inicial, podemos utilizar a tabela `ponto`, criada pelos comandos abaixo:

```
ponto <- tibble(
  usuario = "Ana",
  dia = c(1, 1, 1, 2, 2),
  hora = c(14, 14, 18, 8, 13),
  minuto = c(30, 30, 50, 0, 30),
  tipo = c("E", "E", "S", "E", "E"),
  mes = 3,
  ano = 2020
)

ponto

## # A tibble: 5 x 7
##   usuario  dia hora minuto tipo  mes  ano
##   <chr> <dbl> <dbl> <dbl> <chr> <dbl> <dbl>
```



```
##   <chr>   <dbl> <dbl>   <dbl> <chr>   <dbl> <dbl>
## 1 Ana      1    14     30 E      3  2020
## 2 Ana      1    14     30 E      3  2020
## 3 Ana      1    18     50 S      3  2020
## 4 Ana      2     8      0 E      3  2020
## 5 Ana      2    13     30 E      3  2020
```

Inicialmente, a função `distinct()` funciona da mesma maneira que a função `unique()`. Porém, a função `unique()` pode ser aplicada em praticamente qualquer tipo de estrutura, além do tipo de estrutura adotado em seu resultado, variar em diversas aplicações. Enquanto isso, a função `distinct()` (assim como as demais funções do pacote `dplyr`) irá sempre aceitar um `data.frame` como *input* e gerar um novo `data.frame` como *output*. Logo, se aplicarmos `distinct()` sobre a tabela `ponto`, temos o seguinte resultado:

```
ponto_dis <- distinct(ponto)

ponto_dis

## # A tibble: 4 x 7
##   usuario   dia hora minuto tipo   mes   ano
##   <chr>   <dbl> <dbl>   <dbl> <chr> <dbl> <dbl>
## 1 Ana      1    14     30 E      3  2020
## 2 Ana      1    18     50 S      3  2020
## 3 Ana      2     8      0 E      3  2020
## 4 Ana      2    13     30 E      3  2020
```

Repare pelo produto acima, que a função `distinct()` eliminou a segunda linha da tabela `ponto`, pois essa era uma duplicata da primeira linha. Para mais, a função `distinct()` nos permite aplicar a função sobre colunas específicas do `data.frame` em questão. No exemplo acima, nós omitimos essa funcionalidade, e pedimos para que a função `distinct()` fosse aplicada sobre a toda a tabela. Isso significa, que ao não definirmos uma coluna ou um conjunto de colunas em particular, `distinct()` vai utilizar a combinação dos valores de todas as colunas para determinar os valores únicos presentes em sua tabela e, portanto, eliminar os valores duplicados segundo essa abordagem.

Como exemplo, podemos aplicar a função sobre as colunas `usuario` e `tipo`. Dessa forma, `distinct()` nos retorna um novo `data.frame` contendo os valores únicos presentes nessas colunas. No entanto, perceba que um efeito colateral foi gerado, pois nós perdemos todas as demais colunas da tabela `ponto` durante o processo. Isso ocorre em todas as ocasiões em que listamos uma combinação de colunas em `distinct()`. Para evitar esse comportamento, você pode definir o argumento `.keep_all` para `TRUE`, como no exemplo abaixo.

```
ponto_dis <- distinct(ponto, usuario, tipo)

ponto_dis
```

```
## # A tibble: 2 x 2
##   usuario tipo
##   <chr>   <chr>
## 1 Ana     E
## 2 Ana     S

ponto_dis <- distinct(ponto, usuario, tipo, .keep_all = TRUE)

ponto_dis
```

```
## # A tibble: 2 x 7
##   usuario dia hora minuto tipo mes ano
##   <chr>   <dbl> <dbl> <dbl> <chr> <dbl> <dbl>
## 1 Ana     1    14    30 E      3  2020
## 2 Ana     1    18    50 S      3  2020
```

Com isso, se desejamos eliminar os valores duplicados em nossas tabelas, podemos rapidamente aplicar a função `distinct()` sobre toda a tabela. Contudo, haverá momentos em que combinações específicas de colunas devem ser utilizadas para determinarmos as observações únicas da tabela, ao invés de todas as colunas disponíveis. Para isso, você deve listar os nomes das colunas a serem utilizadas pela função `distinct()` neste processo. Além disso, você geralmente vai desejar utilizar a configuração `.keep_all = TRUE` durante essa situação, com o objetivo de conservar as demais colunas da tabela no resultado de `distinct()`.

Ademais, lembre-se que você pode utilizar a função `across()` para ter acesso aos mecanismos de seleção de `select()`, para definir o conjunto de colunas a ser empregado por `distinct()`. Por exemplo, eu posso encontrar todos os valores únicos criados pela combinação entre as colunas `dia` e `tipo`, por meio do seguinte comando:

```
distinct(ponto, across(c(2, 5)), .keep_all = TRUE)

## # A tibble: 3 x 7
##   usuario dia hora minuto tipo mes ano
##   <chr>   <dbl> <dbl> <dbl> <chr> <dbl> <dbl>
## 1 Ana     1    14    30 E      3  2020
## 2 Ana     1    18    50 S      3  2020
## 3 Ana     2     8     0 E      3  2020
```

4.11 Combinando tabelas com `bind_cols()` e `bind_rows()`

Os pacotes básicos do R oferecem as funções `rbind()` e `cbind()`, que lhe permite combinar objetos. Porém, o pacote `dplyr` oferece implementações mais rápidas e completas desse mecanismo, através das funções `bind_cols()` e `bind_rows()`. Do mesmo modo que as demais funções do pacote,

`bind_cols()` e `bind_rows()` aceitam um conjunto de `data.frame`'s como *input*, e lhe retornam um novo `data.frame` como *output*.

Como exemplo inicial, suponha que você possua o conjunto de tabelas abaixo. Essas tabelas contêm dados das vendas de três lojas diferentes.

```
savassi <- tibble(  
  dia = as.Date(c("2020-03-01", "2020-03-02", "2020-03-03",  
                  "2020-03-04")),  
  produtoid = c("10241", "10241", "10032", "15280"),  
  loja = "Savassi",  
  unidades = c(1, 2, 1, 1),  
  valor = c(15.5, 31, 12.4, 16.7)  
)  
  
prado <- tibble(  
  dia = as.Date(c("2020-03-10", "2020-03-11", "2020-03-12")),  
  produtoid = c("15280", "10032", "10032"),  
  loja = "Prado",  
  unidades = c(3, 4, 2),  
  valor = c(50.1, 49.6, 24.8)  
)  
  
centro <- tibble(  
  dia = as.Date(c("2020-03-07", "2020-03-10", "2020-03-12")),  
  produtoid = c("15280", "15280", "15280"),  
  loja = "Centro",  
  unidades = c(5, 1, 1),  
  valor = c(83.5, 16.7, 16.7)  
)
```

Supondo que você seja um analista da empresa dona dessas lojas, e que foi delegado a você, a tarefa de analisar os dados dessas tabelas, você terá no mínimo o triplo de trabalho, caso mantenha essas tabelas separadas. Pois cada etapa de sua análise teria de ser replicar em três lugares diferentes. Por isso, a melhor opção é reunir essas tabelas em um lugar só. Pois dessa maneira, você precisa aplicar as suas operações em um único lugar.

Ou seja, a motivação para o uso das funções `bind_rows()` e `bind_cols()`, surge em geral, a partir da dificuldade que temos em aplicar a mesma função em diversos pontos de nosso trabalho, além da manutenção e monitoramento dos resultados gerados em cada um desses pontos envolvidos nesse serviço. Portanto, se você possui um grande conjunto de tabelas, que são semelhantes entre si, e você precisa aplicar os mesmos passos sobre cada uma delas, é interessante que você tente juntar essas tabelas em uma só. Dessa maneira, você pode direcionar o seu foco e as suas energias para um só local.

Como as tabelas `savassi`, `prado` e `centro` possuem as mesmas colunas, faz mais sentido unirmos

as linhas de cada tabela para formarmos a nossa tabela única. Para isso, basta listarmos essas tabelas dentro da função `bind_rows()`, como demonstrado abaixo. Uma outra opção, seria provermos uma lista de `data.frame`'s à função, o que também está demonstrado abaixo:

```
bind_rows(savassi, prado, centro)
```

```
## # A tibble: 10 x 5
##   dia      produtoid loja  unidades valor
##   <date>    <chr>    <chr>    <dbl> <dbl>
## 1 2020-03-01 10241    Savassi      1  15.5
## 2 2020-03-02 10241    Savassi      2   31
## 3 2020-03-03 10032    Savassi      1  12.4
## 4 2020-03-04 15280    Savassi      1  16.7
## 5 2020-03-10 15280    Prado        3  50.1
## 6 2020-03-11 10032    Prado        4  49.6
## 7 2020-03-12 10032    Prado        2  24.8
## 8 2020-03-07 15280    Centro       5  83.5
## 9 2020-03-10 15280    Centro       1  16.7
## 10 2020-03-12 15280    Centro       1  16.7
```

```
## -----
## Uma alternativa seria fornecermos uma lista
## contendo as tabelas a serem unidas:
```

```
lista <- list(savassi, prado, centro)
```

```
bind_rows(lista)
```

Portanto, ao unir as linhas de cada tabela, a função `bind_rows()` está de certa forma “empilhando” uma tabela em cima da outra. Mas para que este tipo de operação ocorra de maneira adequada, é **importante que as colunas de todas as tabelas estejam nomeadas igualmente**. Dito de outra forma, as tabelas envolvidas nesse cálculo, devem ser formadas pelo mesmo grupo de colunas. Essas colunas podem se encontrar em ordens diferentes ao longo das tabelas, mas elas precisam necessariamente estar nomeadas da mesma maneira. Caso alguma coluna em pelo menos uma das tabelas possua um nome diferente de seus pares, a função vai alocar os seus valores em uma coluna separada das demais, e isso geralmente não é o que você deseja.

```
colnames(centro)[2:3] <- c("ProdutoID", "Loja")
```

```
bind_rows(savassi, prado, centro)
```

```
## # A tibble: 10 x 7
##   dia      produtoid loja  unidades valor ProdutoID Loja
##   <date>    <chr>    <chr>    <dbl> <dbl> <chr>    <chr>
```

```
## 1 2020-03-01 10241 Savassi 1 15.5 <NA> <NA>
## 2 2020-03-02 10241 Savassi 2 31 <NA> <NA>
## 3 2020-03-03 10032 Savassi 1 12.4 <NA> <NA>
## 4 2020-03-04 15280 Savassi 1 16.7 <NA> <NA>
## 5 2020-03-10 15280 Prado 3 50.1 <NA> <NA>
## 6 2020-03-11 10032 Prado 4 49.6 <NA> <NA>
## 7 2020-03-12 10032 Prado 2 24.8 <NA> <NA>
## 8 2020-03-07 <NA> <NA> 5 83.5 15280 Centro
## 9 2020-03-10 <NA> <NA> 1 16.7 15280 Centro
## 10 2020-03-12 <NA> <NA> 1 16.7 15280 Centro
```

Por outro lado, quando estamos planejando unir tabelas a partir de suas colunas, a nossa preocupação principal deve ser com o número de linhas de cada tabela. Com isso, quando utilizar a função `bind_cols()`, é **essencial que as tabelas envolvidas possuam exatamente o mesmo número de linhas**. Ou seja, no caso da função `bind_cols()`, é primordial que as tabelas fornecidas à função, possuam o mesmo número de linhas, pois caso contrário, um erro será acionado pela função, e você não poderá prosseguir.

Tendo esse ponto em mente, você utiliza a função `bind_cols()` do mesmo modo que a função `bind_rows()`. Basta listar as tabelas a serem unidas dentro da função, ou fornecer uma lista contendo os `data.frame`'s a serem fundidos. Veja abaixo, um exemplo com as tabelas `tab1` e `tab2`.

```
tab1 <- tibble(
  dia = 1:5,
  valor = round(rnorm(5), 2)
)

tab2 <- tibble(
  id = c("104", "104", "105", "106", "106"),
  nome = "Ana"
)

bind_cols(tab1, tab2)

## # A tibble: 5 x 4
##   dia valor id   nome
##   <int> <dbl> <chr> <chr>
## 1     1  0.35 104   Ana
## 2     2 -0.17 104   Ana
## 3     3 -0.59 105   Ana
## 4     4 -1.33 106   Ana
## 5     5 -1.1  106   Ana

## -----
## Uma alternativa seria fornecermos uma lista
```

```
## contendo as tabelas a serem unidas:
```

```
lista <- list(tab1, tab2)
```

```
bind_cols(lista)
```

Exercícios

Uma excelente forma de exercitar os conhecimentos adquiridos nesse capítulo é simplesmente brincar com bases de dados diferentes! Simplesmente, brinque! Tente encontrar fatos curiosos sobre cada base de dados. Faça perguntas (quantas pessoas se encaixam nessa categoria? Quantas mensagens foram enviadas durante esse dia? Qual é o custo médio de um curso de Medicina nos EUA?) e tente respondê-las com as funções que você descobriu nesse capítulo.

Por isso, a maioria dos exercícios a seguir são exercícios práticos, que pedem por uma informação específica a respeito dos dados contidos em uma determinada tabela (isto é, um `data.frame`). Para chegar a essa informação, você pode utilizar as funções do pacote `dplyr`. O código necessário para ter acesso a cada uma dessas tabelas será fornecido em cada questão.

O projeto [TidyTuesday](#) é um ótimo lugar para encontrarmos diferentes bases de dados. Pois todas essas bases estão hospedadas na pasta do projeto no GitHub, e grande parte delas estão guardadas em arquivos de texto (`.txt`, `.csv`, `.fwf`, etc.), os quais podemos ler e importar diretamente da página do GitHub para o R, sem a necessidade de baixar arquivos manualmente. Em resumo, o Tidy Tuesday é um projeto onde os integrantes disponibilizam toda semana, uma base de dados diferente. Qualquer pessoa pode submeter uma base de dados para o projeto, incluindo ou não, artigos e materiais que possam instruir os usuários sobre como analisar e compreender os dados contidos nessa base.

Exercício 1

Antes de partirmos para as questões práticas, vamos exercitar o seu conhecimento sobre cada função mostrada nesse capítulo. Em cada item abaixo, eu forneço um conjunto de comandos. Cada conjunto inclui funções do pacote `dplyr` (como `filter()`, `mutate()`, `group_by()`, etc.) e uma tabela específica logo no início dessa cadeia de comandos, além do operador *pipe* (`%>%`) conectando cada uma das funções aplicadas. Seu trabalho é ler esse conjunto de comandos, e descrever mentalmente (ou escrever em algum papel) o que cada uma das funções aplicadas está fazendo nessa cadeia. Em outras palavras, seu objetivo é descrever o papel que cada função desempenha nessa cadeia de comandos.

A) Descreva os comandos abaixo:

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

B) Descreva os comandos abaixo:

```
vec <- c("species", "homeworld", "films", "vehicles", "starships")  
  
starwars %>%  
  select(-all_of(vec)) %>%  
  group_by(sex) %>%  
  summarise(peso_medio = mean(mass, na.rm = TRUE))
```

c) Descreva os comandos abaixo:

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

Exercício 2

Vamos começar pela base `tuition_income.csv`, referente a semana 11 do TidyTuesday em 2020. Com os comandos abaixo, você pode rapidamente importar essa base de dados para o seu R. Os dados contidos nessa base, descrevem um conjunto de universidades dos Estados Unidos durante o período de 2010 a 2018, e oferecendo informações como: nome da faculdade/universidade (`name`); estado em que ela se encontra (`state`); preço ou custo total (em dólares) exigido pela graduação na instituição (`total_price`); ano ao qual os valores se referem (`year`); fica localizada dentro do campus ou fora dele? (`campus`); custo total líquido (custo total menos bolsas de auxílio e prêmios) pago pela graduação na instituição (`net_cost`).

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/rfordatascience/"
arquivo <- "tidytuesday/master/data/2020/2020-03-10/tuition_income.csv"

dados <- read_csv(paste0(github, arquivo))
```

- A) Com esses dados em mão, tente descobrir as 10 universidades que sofreram os maiores aumentos de preços durante o período descrito na base.
- B) Dado que você descubra a universidade que sofreu o maior aumento de preço dentre as 10 universidades descritas no item anterior, procure pelos dados dessa universidade ao longo da base. Com esses dados, discuta o momento em que houve a variação. Não tem resposta certa ou errada, apenas encontre os dados dessa universidade na base, e dê sua opinião sobre tamanha variação no preço dessa universidade.

Exercício 3

Caso você tenha chamado pelo pacote dplyr com sucesso em sua sessão, através do comando `library()`, você tem acesso à tabela `starwars`, que é utilizada nessa questão. Tendo a tabela `starwars` em mãos, da coluna 1 até a coluna 11, descubra qual a coluna do tipo `character` que possui o maior número de valores únicos (ou o maior número de “grupos”) ao longo da base.

```
starwars

## # A tibble: 87 x 14
##   name      height mass hair_color skin_color eye_color birth_year sex
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>      <dbl> <chr>
## 1 Luke Sky~    172    77 blond      fair        blue         19 male
## 2 C-3PO       167    75 <NA>      gold        yellow       112 none
## 3 R2-D2        96    32 <NA>      white, bl~ red         33 none
## 4 Darth Va~   202   136 none      white      yellow      41.9 male
## 5 Leia Org~   150    49 brown     light      brown        19 fema~
## 6 Owen Lars   178   120 brown, grey light      blue         52 male
## 7 Beru Whi~   165    75 brown     light      blue         47 fema~
## 8 R5-D4        97    32 <NA>      white, red red         NA none
## 9 Biggs Da~   183    84 black     light      brown        24 male
## 10 Obi-Wan ~   182    77 auburn, wh~ fair        blue-gray    57 male
## # ... with 77 more rows, and 6 more variables: gender <chr>,
## #   homeworld <chr>, species <chr>, films <list>, vehicles <list>,
## #   starships <list>
```

Exercício 4

Vamos agora, voltar rapidamente para a base de dados `transf` que visitamos ao longo deste capítulo. Lembre-se que você pode importar essa base diretamente para o seu R, ao copiar e colar os comandos abaixo.

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/pedropark99/"
pasta <- "Curso-R/master/Dados/"
arquivo <- "transf_reform.csv"

transf <- read_csv2(paste0(github, pasta, arquivo))
```

- A) Qual é a receita média que o atendente Eduardo realiza com transferências destinadas à Alemanha?
- B) Qual é o país de destino com o qual a atendente Ana mais trabalha?
- C) Descubra quais foram as últimas transferências executadas por cada um dos 8 atendentes presentes em `transf`. Lembre-se que a coluna `Data` fornece o ponto do tempo em que a transferência foi executada.

Capítulo 5

Funções e Loops no R

5.1 Introdução

Neste capítulo, veremos apenas uma introdução de como você pode criar as suas próprias funções no R, e automatizar alguns passos utilizando *loop*'s. É importante frisar que no início pode ser bem difícil de implementar a sua função. Isso ocorre principalmente, porque as funções quando executadas, rodam em um ambiente diferente do seu.

Você talvez tenha ficado confuso com essa afirmativa, se perguntando: “O que diabos você quer dizer com um ambiente diferente?”. O R não só é uma linguagem que trabalha com objetos, mas ele também é uma linguagem que trabalha com objetos que estão guardados em ambientes específicos. Os ambientes no R são comumente chamados por *environments*. Vamos descrevê-los em mais detalhes na próxima seção.

Tudo o que você precisa entender agora, é que todas as operações que uma função realiza, ocorrem de forma implícita, em um local onde você não consegue ver cada uma dessas operações. É essa característica que torna bem difícil no início, a criação de funções pelo usuário. Ao tentar implementar a sua função pela primeira vez, você vai enfrentar quase sempre, algum erro. E pelo fato de você não conseguir acompanhar cálculo por cálculo da função, quando um erro aparece, a principal pergunta que você se faz é: “De onde este erro está vindo? Em que lugar da minha função ele ocorre?”

Tendo essas considerações em mente, vou mostrar aqui como você pode começar a montar as suas próprias funções, dando algumas dicas de como enfrentar erros, e quais são as formas de organizá-las, para que você não se perca no meio do processo.

5.2 Noções básicas de *environments*

O R é não apenas uma linguagem que trabalha com objetos, mas também é uma linguagem que trabalha com objetos que estão contidos (ou guardados) em certos *environments*. Um *environment*

(ou ambiente) no R, é parecido com uma lista nomeada, **onde cada nome mantido nessa lista é único**. Sendo que esses nomes dispostos em uma espécie de lista, nada mais são do que os nomes dos objetos que estão guardados e disponíveis nesse respectivo *environment*.

5.2.1 O *environment* global

Toda vez que você inicia a sua sessão no R, você está trabalhando com um *environment* que chamamos de *global environment*, ou ambiente global. Logo, todos os seus objetos que você cria em sua sessão, são guardados no *global environment*. Você pode se referir a esse *environment* através da função `globalenv()`. Por exemplo, eu posso usar a função `ls()` para listar os nomes de todos os objetos que estão disponíveis especificamente no *global environment*, como no exemplo abaixo. No R, o endereço desse *environment* é referenciado como `R_GlobalEnv`.

```
# Iniciei uma nova sessão no R
```

```
a <- 1  
b <- 2
```

```
ls(envir = globalenv())
```

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

Portanto, um *environment* é uma espécie de caixa, ou como um espaço reservado para guardar um certo conjunto de objetos. O seu *global environment* é um desses *environments*, onde ficam todos os seus objetos que você normalmente cria em sua sessão. Porém, todas as vezes que você está trabalhando no R, há diversos outros *environments* ativos. O motivo, ou a razão principal para a existência dessas estruturas, está na forma como a linguagem R procura pelos objetos que você pede a ela. Ou seja, como é destacado por [Wickham \(2015a\)](#) os *environments* são a estrutura que sustentam as regras de *scoping*, ou as regras de “busca” da linguagem R. Mas pelo fato desse ser um assunto mais avançado da linguagem, isto não será abordado aqui. Você pode consultar [Wickham \(2015a\)](#) para mais detalhes.

Tudo o que eu quero destacar nessa seção a respeito de *environments*, é o fato de que todo objeto está ligado a um certo *environment*. Logo toda função (lembre-se que tudo no R são objetos) possui o seu *environment*, e por isso, você terá muitas ocasiões em que você terá de acessar uma função através de seu respectivo *environment*, ao invés de simplesmente chamar pelo nome dessa função no console.

Por essas razões, podemos entender que um dos principais papéis desempenhados pelos *environments* no R, é o de adicionar uma nova camada de identificação de objetos. Se antes o R identificava os valores contidos em objetos, através do nome desse objeto que está conectado a esses valores, com o uso de *environments*, o R agora pode identificar diferentes valores ou diferentes objetos, através do nome desse objeto e do *environment* ao qual ele pertence.

Pense por exemplo, no objeto `LETTERS`. Esse objeto está disponível toda vez que você inicia a sua sessão no R, pois ele se encontra no *environment* base (que faz parte de um dos pacotes básicos da

linguagem, que sempre são carregados para a sua sessão). Nós podemos identificar o *environment* ao qual um objeto pertence, através da função `where()` do pacote `pryr`. Perceba que esse objeto, contém apenas as letras do alfabeto em maiúsculo.

```
LETTERS
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
pryr::where("LETTERS")
```

```
## <environment: base>
```

Portanto, mesmo que eu crie um objeto em minha sessão, ou em outras palavras, um objeto em meu *global environment* chamado `LETTERS`, o R ainda será capaz de diferenciar esses dois objetos denominados `LETTERS`, através do *environment* ao qual eles pertencem. Após criarmos um novo `LETTERS`, se eu chamar por este objeto no console, o resultado será o valor contido no objeto `LETTERS` do meu *global environment*. Isso ocorre, pois o R irá sempre procurar primeiro por um objeto em seu *global environment*. Depois ele irá procurar em outros *environments* pelo objeto ao qual você requisitou. Esse caminho de *environments* pelo qual o R percorre durante sua procura, é comumente chamado por *search path* (ou “caminho de busca”). Para acessarmos o valor do objeto `LETTERS` original, podemos utilizar a função `get()`, que possui um argumento (`envir`) onde podemos definir qual o *environment* em que o R deve procurar pelo objeto. Podemos ver esse problema, através de uma representação visual como a da figura 5.1.

```
env <- pryr::where("LETTERS")
```

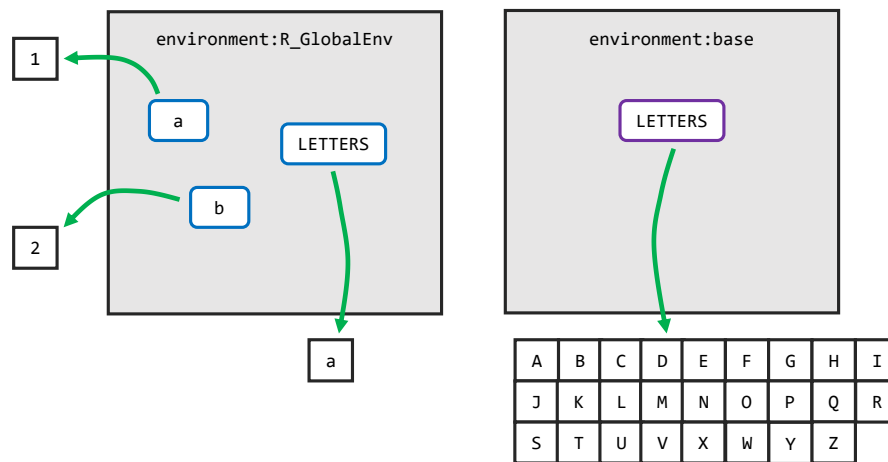
```
LETTERS <- "a"
```

```
LETTERS
```

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

```
get("LETTERS", envir = env)
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Figura 5.1: Representação de environments

Fonte: Elaboração própria do autor.

5.2.2 Os *environments* de pacotes

Portanto, o trabalho de um *environment* no R, é o de vincular, ou de associar um conjunto de nomes (os nomes dos objetos), a seus respectivos conjuntos de valores (WICKHAM, 2015a, Cáp. 7). Ou seja, um dos principais papéis que um *environment* desempenha no R, é o de organizar um conjunto de objetos, de forma que o R seja capaz de diferenciar dois ou mais objetos com o mesmo nome.

Um bom exemplo disso, é a função `filter()` do pacote `dplyr`, que vimos no capítulo 4. Pois nós temos dentre os pacotes básicos do R, mais especificamente no pacote `stats`, uma outra função também chamada `filter()`. Por isso, sempre que chamamos pelo pacote `dplyr` através de `library()`, a seguinte mensagem aparece, nos informando que há um choque entre as duas funções.

```
library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats' :
```

```
filter, lag
```

Essa mensagem está nos informando, que o pacote `dplyr` possui funções com os mesmos nomes das funções `filter()` e `lag()` do pacote `stats`, e por isso, ela estaria “escondendo” esses objetos de forma a evitar conflitos. Isso significa, que após carregarmos o pacote `dplyr`, e ele “esconder”

essas funções, se nós chamarmos pela função `filter()` no console, estaremos utilizando a função do pacote `dplyr`, e não a função do pacote `stats`.

Figura 5.2: Ambientes de pacotes no R



Fonte: Elaboração própria do autor.

Portanto, essas duas funções `filter()`, são funções diferentes, que servem para propósitos diferentes. O único fator que permite ao R, diferenciar essas funções uma da outra, é o fato de que elas se encontram em ambientes, ou *environments* diferentes, como demonstrado na figura 5.2. Isso significa que todas as funções, possuem um *environment* ao qual estão associadas. Por isso, nós podemos diferenciar as funções `filter()` e `lag()` de ambos os pacotes (`dplyr` e `stats`), através do nome do *environment* ao qual essas funções pertencem.

Pacotes são um caso especial, pois eles contêm dois *environments* diferentes que se relacionam entre si. Um é o *environment* propriamente dito do pacote, que contém os seus respectivos objetos, e um outro comumente chamada de *namespace*. Essa diferenciação só será útil na prática, quando você estiver desenvolvendo um novo pacote para o R. Logo, não se preocupe em entender agora a diferença entre esses dois espaços. Apenas entenda que pacotes no R, vão além de simples *environments* que contêm os seus próprios objetos e funções, e que estão separados de seu *global environment*.

Tendo isso em mente, quando desejamos utilizar uma função que está em conflito com uma outra função de um outro pacote, nós devemos definir de alguma forma, o *environment* do pacote no qual o R deve procurar pela função que você está chamando. No caso de pacotes, podemos acessar funções definidas em seus respectivos *environments*, ao fornecer o nome do pacote que contém essa função, em seguida, abrir duas vezes dois pontos (: :), e depois, colocar o nome da função que desejamos, como no exemplo abaixo.

```
x <- ts(rnorm(100), start = c(1, 1990), end = c(4, 1998), frequency = 12)
```

```
stats::filter(x, filter = c(0.5, 0.8, 0.2))
```

```
##           Jan           Feb           Mar           Apr           May
## 166
## 167  1.16442920  0.91500252 -0.23768174 -1.44767486 -0.68237859
## 168  0.88258623  0.42219144  0.33977808  1.47751346  1.49561963
## 169  0.84370768  0.50916936  0.49375355  0.23956192  0.17568137
## 170  1.04628841 -1.13401257 -1.25400780  1.19311175  0.79144899
##           Jun           Jul           Aug           Sep           Oct
## 166
## 167 -0.25496455 -0.63263452 -0.10640841 -0.52051165 -0.82019345
## 168  0.56135033  0.51422710 -0.17227866 -1.18997169  0.02303573
## 169 -0.31809494  0.43692344  2.11986556  0.99620987  0.56587068
## 170           NA
##           Nov           Dec
## 166  0.53303165  0.94640945
## 167 -0.82640133  0.09656562
## 168  1.79113804  1.46001721
## 169  0.60644086  0.59635066
## 170
```

5.2.3 O *environment* de execução de uma função

Toda função no R, possui o que nós chamamos de *function environment*, que corresponde ao *environment* no qual elas foram criadas. No exemplo abaixo, podemos ver esses *environments* pertencentes às funções `filter()` do pacote `dplyr`, e `seq()` do pacote `base`. Como um outro exemplo, caso eu crie uma nova função em minha sessão, perceba que o *environment* a qual ela pertence, se trata justamente do *global environment* (`R_GlobalEnv`).

```
rlang::fn_env(dplyr::filter)

## <environment: namespace:dplyr>

rlang::fn_env(seq)

## <environment: namespace:base>

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

rlang::fn_env(soma)

## <environment: 0x000001cbd11f1bd0>
```


Porém, as funções também possuem o que chamamos de *environment* de execução, que se trata do *environment* no qual os seus cálculos são executados. Ou seja, sempre que você executa uma função, os cálculos realizados por essa função são feitos em um *environment* separado do seu *global environment*. Como consequência, você não consegue visualizar o resultado de cada cálculo ou passo executado por essa função, pois essas etapas estão sendo realizadas em um local distante do *environment* no qual você se encontra. É dessa característica que surge uma das principais dificuldades em se criar a sua própria função. Pois já que você não pode visualizar os resultados de cada cálculo aplicado pela função, você terá que prever quais seriam as possibilidades para cada cálculo. Isso exige de você muita experiência com a linguagem, e nem sempre essa experiência será suficiente para representar a realidade com precisão.

No exemplo abaixo, estou criando uma função `f_env`, que nos retorna justamente o endereço do *environment* no qual essa função executou a soma entre 4 e 5. Perceba que a cada momento em que eu executo essa função, ela me retorna um endereço diferente. Logo, esses *environments* de execução são temporários (ou efêmeros se preferir), e utilizados uma única vez pela função.

```
f_env <- function(){  
  soma <- 4 + 5  
  
  env <- environment()  
  
  return(env)  
}  
  
f_env()  
  
## <environment: 0x000001cbcd288478>  
  
f_env()  
  
## <environment: 0x000001cbcd100978>
```

O motivo principal pelo qual esses *environments* de execução existem, é pelo fato de que nós em geral, não queremos que a função mexa ou altere os nossos objetos salvos em nosso *global environment*. Veja por exemplo, a função `norm` que busca normalizar uma variável numérica qualquer. Nessa função, três objetos são criados. Um contendo a média de `x` (`media`), outro possuindo o desvio-padrão da mesma variável (`desvio_pad`), e um último contendo os valores já normalizados (`normalizado`).

```
norm <- function(x){  
  media <- mean(x)  
  desvio_pad <- sd(x)
```

```

normalizado <- (x - media)/desvio_pad

  return(normalizado)
}

y <- rnorm(50)

norm(y)

## [1] -1.29746780  0.26684044 -0.07699576 -0.35852807 -0.19292098
## [6]  1.62576071 -0.90452041  0.14979621 -2.13009149 -0.31372789
## [11] -0.13906998  1.92566364 -0.60455011 -0.51378355 -1.04466154
## [16]  0.54906817  1.53341938 -0.84350886 -0.06271653  1.74782820
## [21] -0.32396058  1.05999917  0.76093273 -0.95147507  1.59686409
## [26]  1.17079379 -0.12584004 -0.54592611 -0.02704583  1.18987831
## [31] -0.20267237 -1.52616027  0.22302469  1.14728340 -0.80368602
## [36]  0.12703241 -0.86741299  0.48541843  0.23353726  0.33337287
## [41] -0.26134945 -0.34928558  1.24785307  0.47362971  0.17853930
## [46] -2.87897520 -0.92139025  1.13819276 -0.04361194 -0.85339411

```

Pelo fato de que a função `norm` executa os seus cálculos, e portanto, cria os seus objetos em um ambiente separado, mesmo que eu tenha objetos em meu *global environment* que se chamam `media`, ou `desvio_pad`, ou `normalizado`, em nenhum momento a função `norm` irá afetá-los. A função `norm` pode até utilizar-se de objetos que estão guardados em meu *global environment* para realizar os seus cálculos, mas em nenhum momento ela irá alterar o valor dessas variáveis em meu *global environment*, a menos que eu peça explicitamente para que ela realize tal alteração. Isso algo que eu não pretendo mostrar aqui, pois exige de você leitor, um conhecimento e experiência maiores com os *environments* do R.

5.3 Uma introdução teórica às funções no R

As funções, lhe permitem automatizar tarefas, em uma forma mais intuitiva e poderosa, do que uma simples estratégia de copiar-colar (WICKHAM; GROLEMUND, 2017). Portanto, sempre que houver algum tipo de repetição em seu trabalho, você pode utilizar uma função, em conjunto com um *loop*, para automatizar esse processo. Em outras palavras, uma função lhe permite com apenas um comando, aplicar várias outras funções e processos diferentes de uma vez só. Dessa forma, fica mais fácil aplicar repetidamente o mesmo conjunto de processos e comandos sobre os seus dados.

Vamos supor exemplo, que você seja o dono de 4 lojas de doces caseiros, e que essas lojas estão localizadas na cidade de Belo Horizonte, mais especificamente nos bairros Barro Preto, Savassi, Centro e Padre Eustáquio. Toda semana, o gerente de cada loja, te envia uma planilha contendo as vendas diárias de cada produto em sua loja. Você como dono dessas lojas, deseja sempre calcular algumas estatísticas para acompanhar as vendas de suas empresas.

Porém, são 4 planilhas enviadas toda semana, e se você tem que calcular as mesmas estatísticas toda semana, porque não criar uma função que já executa esses cálculos por você? Dessa forma, você consegue calcular todas as suas estatísticas com apenas um comando, economizando tempo e esforço na manutenção dessas estatísticas.

Vamos supor, que as planilhas de cada loja, assumem a seguinte estrutura abaixo. Como exemplo, vamos utilizar a planilha referente a loja localizada no bairro Savassi:

```
savassi
```

```
## # A tibble: 785 x 6
##   dia   mes   ano vendedor produtoid valor
##   <int> <int> <int> <chr>    <chr>    <dbl>
## 1     1     4  2020 Márcia   23010     4.1
## 2     1     4  2020 Ana     10014     7.89
## 3     1     4  2020 Ana     10115    15.4
## 4     1     4  2020 Márcia   53200    11.2
## 5     1     4  2020 Ana     10014     7.89
## 6     1     4  2020 Nathália 53200    11.2
## 7     1     4  2020 Nathália 10014     7.89
## 8     1     4  2020 Márcia   10014     7.89
## 9     1     4  2020 Nathália 53200    11.2
## 10    1     4  2020 Ana     10115    15.4
## # ... with 775 more rows
```

Portanto, cada planilha, apresenta cada venda que ocorreu durante a semana em uma determinada loja, e inclui informações como: o dia em que a venda ocorreu, o vendedor que realizou a venda, o valor da venda, e o código de identificação do produto que foi vendido. Vamos supor, que você esteja interessado em calcular: a receita total diária da loja, o número de vendas diárias de cada produto, e o lucro total dessa semana na loja de referência da planilha. Para isso, precisaríamos dos seguintes comandos:

```
library(dplyr)
library(magrittr)

receita_por_dia <- as.vector(rowsum(savassi$valor, savassi$dia))
names(receita_por_dia) <- paste("Dia", unique(savassi$dia))

vendas_produto <- savassi %>%
  group_by(dia, mes, produtoid) %>%
  summarise(
    receita = sum(valor),
    n_vendas = n()
  ) %>%
```

```
ungroup()

custo <- c("23010" = 1.5, "10014" = 5.43, "10115" = 11, "53200" = 8.9)

vendas_produto <- vendas_produto %>%
  mutate(
    custo_t = custo[vendas_produto$produtoid] * n_vendas,
    lucro = receita - custo_t
  )

lucro_total <- sum(vendas_produto$lucro)
```

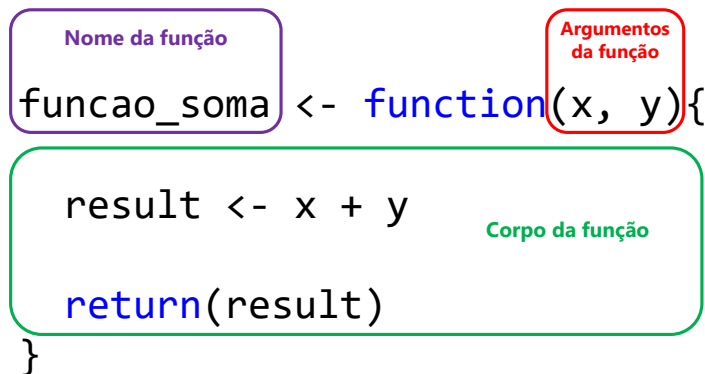
É um conjunto considerável de comandos, e você terá que replicá-los para outras 3 planilhas. Você pode criar uma função, ou um conjunto de funções, com o objetivo de facilitar esse processo de aplicação desses comandos sobre cada planilha. Porém, como temos ao menos três estatísticas diferentes (receita total diária; número de vendas diárias atingidas em cada produto; lucro total da semana), seria mais adequado, criarmos uma função especializada para cada uma dessas estatísticas, e em seguida, criar uma quarta função que será a nossa função principal, que ficará responsável por aplicar essas três funções especializadas que criamos, sobre as tabelas de vendas de cada loja.

Para iniciarmos uma nova função no R, nós precisamos utilizar a palavra chave *function*. Dito de outra forma, essa palavra chave indica ao R, que os próximos comandos, representam os componentes da nossa nova função. Toda função no R, possui três partes principais, mostradas na figura abaixo. Primeiro, temos o nome da função, que corresponde ao nome do objeto em que você está salvando a sua função. Segundo, temos os argumentos, ou os parâmetros dessa função, que são definidos dentro de um par de parênteses posicionados ao lado da palavra chave (*function*). Terceiro, temos o corpo da função, que é definido dentro de um par chaves (*{}*). Dentro desse par de chaves, você deve colocar todos os comandos, ou todos os cálculos que você deseja aplicar sobre os *input*'s da função.

Os *input*'s de uma função, são os objetos, ou os dados que você fornece à função para ela trabalhar. Esses *input*'s são geralmente fornecidos, ao conectarmos um objeto a um dos argumentos da função. Porém, esses *input*'s não precisam estar conectados a algum argumento, nem precisam fazer parte de seu *global environment*.

Toda função no R, possui ainda um quarto componente, que se trata do *environment* ao qual ela pertence, que definimos na seção “O *environment* de execução de uma função”. Esse componente é automaticamente definido no momento em que você cria a função, e é comumente chamado de *function environment*. Para relembrarmos, esse item se trata apenas do *environment* no qual você está criando essa função. Como na maioria do tempo, você estará trabalhando no seu *global environment*, a maioria das funções que você criar, teram o *global environment* como o seu *function environment*.

Como exemplo, eu posso criar através dos comandos abaixo, uma simples função de soma chamada *funcao_soma()*, que possui dois parâmetros, ou dois argumentos chamados *x* e *y*. O objeto soma que

Figura 5.3: Estrutura de uma função

Fonte: Elaboração própria do autor.

está definido dentro do corpo da função, é criado sempre que você executa a função `funcao_soma()`. Porém, como todas as funções são executadas em um ambiente separado de seu *global environment*, o objeto `soma` é criado e salvo em um ambiente diferente do seu, e por isso, você não possui acesso direto ao objeto `soma` a partir de seu *environment*. Mas você possui acesso indireto ao objeto `soma`, se você pedir a função `funcao_soma()`, que lhe retorne (como resultado da função) o que foi salvo neste objeto, como requisitado no exemplo abaixo pela função `return()`.

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

```
funcao_soma(54, 32)
```

```
## [1] 86
```

Os argumentos de toda função, funcionam como apelidos que auxiliam a função, sobre como e onde ela deve posicionar no corpo da função, os valores que fornecermos a esses argumentos. Ou seja, você não precisa nomear os seus objetos com o mesmo nome presente nos argumentos de uma função. Pois a função utiliza o nome de seus argumentos, apenas como um meio de transportar os seus *input's* para o seu corpo.

Como exemplo, eu possuo abaixo uma função chamada `extrair()`. Pelo corpo da função, podemos identificar que essa função irá extrair os valores de uma coluna chamada `horario`, do objeto que

conectarmos ao argumento `x`, e irá salvar esses valores em um objeto chamado `extr_x`. Por outro lado, também podemos perceber que a função `extrair()` busca aplicar a função `order()` sobre os valores contidos na coluna `valor`, que está presente no objeto que forneceremos ao argumento `y`. Porém, no caso do argumento `z`, a função vai apenas somar os valores (através da função `sum()`) contidos no objeto que oferecermos à esse argumento da função `extrair()`.

```
extrair <- function(x, y, z){  
  extr_x <- x$horario  
  extr_y <- order(y$valor)  
  calc_z <- sum(z)  
}
```

Para acessarmos a função que acabamos de criar, precisamos apenas utilizar o nome do objeto onde salvamos a função e abrirmos um par de parênteses. Logo, se eu quisesse acessar a função `extrair`, eu devo utilizar os comandos abaixo. Lembre-se que ao utilizar uma função, você pode definir um argumento implicitamente, ou explicitamente. Se você não deixa claro a que argumento um objeto está conectado, esse objeto é conectado ao argumento correspondente a sua ordem. Em outras palavras, no exemplo abaixo, o primeiro objeto dentro do parênteses (como o objeto `funcionarios` abaixo), é conectado ao primeiro argumento da função `extrair` (argumento `x`), o segundo objeto (`vendas`), ao segundo argumento (argumento `y`), e assim por diante. Porém, caso você deseja fornecer algum objeto fora da ordem dos argumentos de uma função, você deve definir explicitamente qual o argumento que aquele objeto deve ser direcionado. Para isso, basta igualar o objeto ao nome do argumento desejado da função.

```
extrair(funcionarios, vendas, vendas)
```

```
extrair(funcionarios, z = vendas, y = vendas)
```

5.4 Construindo um conjunto de funções

Vamos retornar ao nosso exemplo, em que você é dono de 4 lojas de doces localizadas na cidade de Belo Horizonte, e você precisa toda semana manter um conjunto de três estatísticas sobre cada loja. Em mais detalhes, precisamos calcular para cada loja, as seguintes estatísticas:

- 1) Receita total diária (queremos identificar se houve alguma variação importante na receita, ao longo da semana).
- 2) Número de vendas diárias de cada produto (queremos saber quais produtos estão bombando).
- 3) Lucro total da semana (nessa semana, entramos no azul? ou no vermelho?).

Eu poderia muito bem criar uma única função que é capaz de calcular todas essas estatísticas de uma vez só. Porém, com o objetivo de incentivar melhores práticas, vou criar um conjunto de 4 funções. Dentre esse conjunto, 3 função serão responsáveis por calcular as estatísticas, e uma quarta função

será responsável por aplicar as três funções anteriores sobre alguma tabela que forneceremos como *input*.

5.4.1 Calculando a receita total diária

Vamos começar, com uma função para calcular a receita total diária de cada loja. Primeiro, precisamos decidir um nome para o nossa função, ou em outras palavras, o nome do objeto em que vamos salvar a nossa função. O nome desse objeto, se torna o nome dessa função. Dessa forma, podemos acessar a função salva no objeto `receita_diaria`, ao abrirmos parênteses após esse nome.

Para calcularmos a receita total diária, precisamos de duas informações: as colunas `valor` e `dia`. A coluna `dia`, define os grupos que queremos utilizar para calcular a nossa receita total. E a coluna `valor`, informa a receita adquirida em cada venda. A função `rowsum()` representa uma solução eficiente para calcularmos essa receita, e possui dois argumentos principais: 1) `x`, o vetor (ou `data.frame`) com os valores a serem somados; 2) `group`, o vetor contendo os valores que apresentam o grupo de cada observação em `x`. Porém, como essa função costuma nos retornar uma matriz como resultado, eu posso aplicar a função `as.vector()` sobre ela, para transformar esse resultado em um vetor.

```
receita_diaria <- function(vl, grp){  
  
  receita <- as.vector(rowsum(x = vl, group = grp))  
}  
  
receita_diaria(savassi$valor, savassi$dia)
```

Como definimos anteriormente, os nomes dos argumentos (`vl` e `grp`) da nossa função, são apenas pronomes, ou apelidos para os *input*'s da nossa função. Esses apelidos, servem apenas para guiar a função, e determinam onde esses *input*'s devem ser posicionados no corpo da função. No exemplo acima, o pedaço `rowsum(x = vl` no corpo da função, indica que o *input* que nós forneceremos ao argumento `vl` da função `receita_diaria()`, deve ser conectado ao argumento `x` da função `rowsum()`.

Você talvez tenha percebido, que ao executarmos a função `receita_diaria()` acima, nenhum resultado foi retornado. Isso ocorre, porque nós não utilizamos no corpo da função, algum comando para retornar o objeto (`receita`) onde o resultado foi salvo. Ou seja, a função aplicou sim os comandos que definimos em seu corpo. Ela apenas não se preocupou em nos retornar o resultado. Para isso, podemos simplesmente chamar pelo nome do objeto, ao final do corpo da função. Ou utilizar uma função como `print()` ou `return()` para nos retornar esse resultado. Eu prefiro utilizar uma função, pois é uma maneira mais formal e clara de se definir o objeto que contém o resultado de sua função. Veja abaixo, que ao colocarmos a função `return()` no corpo, a nossa função passa a nos retornar as receitas totais diárias calculadas.

```
receita_diaria <- function(vl, grp){
```

```

receita <- as.vector(rowsum(x = vl, group = grp))

return(receita)
}

receita_diaria(savassi$valor, savassi$dia)

## [1] 1508.27 1714.14 1336.58 1407.90 1766.22

```

Entretanto, o vetor resultante de `receita_diaria()`, ainda carece de alguma notação, que seja capaz de nos informar o dia que cada valor presente no vetor se refere. Ou seja, o valor 1508,27 se refere a que dia do mês? Dia 01? 02? 03? ... Com isso, eu utilizo a função `names()` sobre o objeto `receita`, para definir um nome para cada um desses valores, contendo o dia a que eles se referem.

```

receita_diaria <- function(vl, grp){

  receita <- as.vector(rowsum(x = vl, group = grp))
  names(receita) <- paste("Dia", unique(grp))

  return(receita)
}

receita_diaria(savassi$valor, savassi$dia)

##   Dia 1   Dia 2   Dia 3   Dia 4   Dia 5
## 1508.27 1714.14 1336.58 1407.90 1766.22

```

5.4.2 Calculando o número de vendas diárias de cada produto

Agora que temos a função `receita_diaria()`, podemos passar agora, para o cálculo do número de vendas diárias atingidas em cada produto. Vou chamar essa função de `produtos_vendas()`, e ela vai possuir apenas um argumento (`dados`), que representa a planilha de cada loja, contendo os dados de cada venda da semana.

Para realizarmos o cálculo desejado, precisamos realizar duas etapas: 1) pegar a tabela que forneceremos como *input* e agrupá-la, ou seja, precisamos definir o grupo da tabela, e para isso, podemos utilizar a função `group_by()`; 2) em seguida, podemos calcular as estatísticas e guardá-las em uma nova tabela. A função `summarise()` é uma boa alternativa para essa etapa. Uma outra etapa opcional, mas não obrigatória nesse cálculo, é a de eliminar a definição dos grupos da tabela, com o objetivo de retornar a tabela calculada, para o estado de uma tabela tradicional. Como eu disse, essa é uma etapa opcional, e que não afeta em nada a nossa tabela em si, estamos apenas “desagrupando” a tabela, ou eliminando a descrição que definia os grupos dessa tabela. Para essa etapa opcional, podemos utilizar a função `ungroup()`.


```

produtos_vendas <- function(dados){

  vendas_produto <- dados %>%
    group_by(dia, produtoid) %>%
    summarise(
      receita = sum(valor),
      n_vendas = n()
    ) %>%
    ungroup()

  return(vendas_produto)
}

produtos_vendas(savassi) %>% print(n = 10)

## # A tibble: 20 x 4
##       dia produtoid receita n_vendas
##   <int> <chr>      <dbl>    <int>
## 1     1  1 10014      339.      43
## 2     1  1 10115      680.      44
## 3     1  1 23010      152.      37
## 4     1  1 53200      338.      30
## 5     2  2 10014      284.      36
## 6     2  2 10115      865.      56
## 7     2  2 23010      160.      39
## 8     2  2 53200      405       36
## 9     3  3 10014      252.      32
## 10    3  3 10115      525.      34
## # ... with 10 more rows

```

5.4.3 Calculando o lucro total

Após criarmos as funções `produtos_vendas()` e `receita_diaria()`, precisamos de uma função para calcularmos o lucro total. Vou dar o nome de `calc_lucro()` a essa função. Sendo que para o cálculo do lucro, precisamos apenas subtrair o custo total da receita total. O cálculo da receita total é simples, precisamos apenas somar os valores dispostos na coluna `valor` da tabela de cada loja.

Já para o custo total, temos uma solução prática através do uso de *subsetting*. Precisamos primeiro criar um vetor contendo os custos de cada produto (vetor `custo`). Em seguida, nós replicamos esse vetor ao longo da tabela de vendas de cada loja, de acordo com cada produto vendido (`custo[x$produtoid]`). Por último, nós somamos os custos desse vetor para chegarmos ao custo total.

```

calc_lucro <- function(x){

  custo <- c("23010" = 1.5, "10014" = 5.43, "10115" = 11, "53200" = 8.9)

```

```
custo_total <- sum(custo[x$produtoid])

receita_total <- sum(x$valor)

lucro_total <- receita_total - custo_total

return(lucro_total)
}

calc_lucro(savassi)

## [1] 2367.09
```

5.4.4 Agrupando essas funções em uma só

Após construirmos todas as três funções que precisamos aplicar, nós podemos criar uma última função, que irá aplicar todas essas três funções de uma vez só. Agora, como podemos retornar múltiplos resultados em uma função? Pois nós temos agora, três resultados diferentes a serem calculados pela função, e nós gostaríamos de ter acesso a cada um desses três resultados. A resposta para essa pergunta, é uma lista. Pois você pode incluir o que você quiser dentro de uma lista, logo, basta colocarmos todos os resultados em uma lista, e pedir a função `return()` que nos retorne essa lista, como resultado da função `calc_stats()` abaixo.

```
calc_stats <- function(tabela){

  receita <- receita_diaria(vl = tabela$valor, grp = tabela$dia)

  lucro_total <- calc_lucro(tabela)

  produtos_vendas <- produtos_vendas(tabela)

  lista <- list(
    receita = receita,
    lucro = lucro_total,
    vendas = produtos_vendas
  )

  return(lista)
}
```

Com a função `calc_stats()`, fica mais simples replicar os cálculos ao longo das tabelas de cada loja, como no exemplo abaixo.

```
savassi_stats <- calc_stats(savassi)
barro_preto_stats <- calc_stats(barro_preto)
padre_eustaquio_stats <- calc_stats(padre_eustaquio)
centro_stats <- calc_stats(centro)
```

5.5 Introduzindo loops

Como o próprio nome dá a entender, um *loop* busca criar um ciclo que se repete em torno de uma operação (ou de um conjunto de operações). Em outras palavras, um *loop* cria uma repetição, e em uma dessas repetições, um mesmo conjunto de operações são aplicadas pelo R. O objetivo de um *loop*, é que a cada repetição, podemos alterar os *input*'s das funções que estão sendo aplicadas nessas operações, e conseqüentemente, os seus resultados.

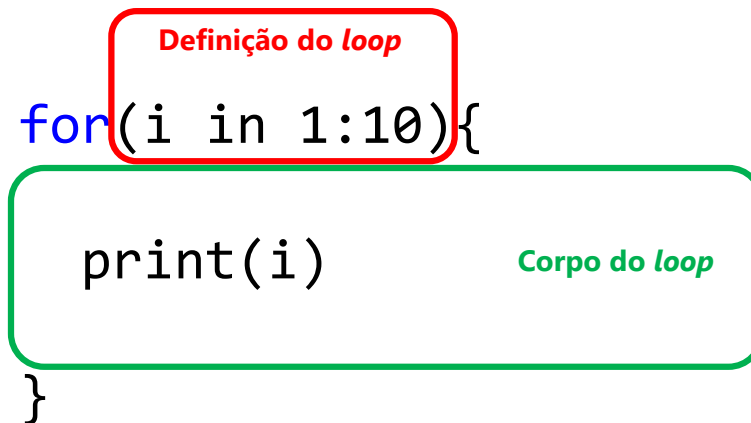
Na seção passada, vimos como podemos criar funções, que representam uma forma concisa de aplicar todos os seus passos e cálculos, com apenas um comando. Agora, com o uso de *loop*'s, podemos aplicar repetidamente uma função ao longo de diversos pontos, ou sobre diversos *input*'s diferentes. Dito de outra forma, com um *loop*, podemos automatizar o processo de aplicação de uma função ao longo de diferentes *input*'s.

Na linguagem R, temos dois tipos, ou duas famílias principais de *loop*'s. Mas aqui vou focar apenas em um desses tipos, que é o *loop* `for()`. A estrutura básica de um `for()` *loop*, se assemelha muito à estrutura de uma função. Porém, diferente de uma função, você não precisa salvar um `for()` *loop* em algum objeto. Um `for()` *loop* é sempre iniciado pela palavra chave `for`. Ao lado dessa palavra chave, devem ser posicionados um par de parênteses, e um par de chaves. Criando assim, a estrutura básica abaixo.

```
for(i in vetor){
  # Corpo do loop
}
```

Um `for()` *loop* possui dois componentes principais: 1) a definição do *loop*, que rege como os índices do *loop* vão variar a cada repetição; 2) e o corpo deste *loop*, que contém todas as funções e operações que serão aplicadas em cada repetição. A definição do *loop* é sempre definida nos parênteses que estão logo em seguida da palavra chave `for`, e será onde você irá definir quantas vezes o *loop* irá repetir, e também, sobre qual conjunto de valores ele vai variar.

Um `for()` *loop* possui um índice, que a cada repetição, assume um valor diferente. Você deve determinar o conjunto de valores que esse índice vai assumir, na definição do *loop*, ao fornecer um vetor (ou uma função que retorne um vetor) contendo esse conjunto de valores, a direita da palavra chave `in`. Sendo que através desse vetor, ou desse conjunto de valores, você está indiretamente determinando quantas vezes o *loop* vai repetir as operações definidas em seu corpo. Ou seja, se o vetor fornecido na definição do *loop*, possui 5 valores diferentes, o *loop* vai gerar 5 repetições, mas se esse vetor possui 10 valores, o *loop* vai gerar 10 repetições, e assim por diante.

Figura 5.4: Estrutura de um loop

Fonte: Elaboração própria do autor.

Já o corpo do *loop*, funciona da mesma forma que o corpo de uma função. Dessa maneira, o corpo de um *loop*, contém todas as funções e transformações que o `for()` *loop* vai aplicar em cada repetição. Com isso, você vai utilizar o índice do *loop*, para variar os input's utilizados pelas funções presentes no corpo desse *loop*, e conseqüentemente, variar os seus resultados gerados a cada repetição do *loop*.

O índice dos *loop*'s mostrados nessa seção, é representado pela letra *i*. Você pode dar o nome que quiser a esse índice, basta substituir o *i* na definição do *loop* pelo nome desejado. No nosso caso aqui, nós optamos por utilizar a letra *i* para nos referir a esse índice, e por isso, em todos os locais do corpo do *loop*, em que a letra *i* aparece sozinha¹, temos um local onde o índice será utilizado. Em outras palavras, se o meu índice é chamado pela letra *i*, e eu possuo uma parte do corpo de meu *loop*, como `sum(i)`, isso significa que a cada repetição do *loop*, a função `sum()` será aplicada sobre o valor que o índice *i* assumir nessa respectiva repetição.

O exemplo mais simples de um *loop*, seria um que mostra cada valor incluso no vetor que eu forneci na definição do *loop*. No exemplo abaixo, estou criando um *loop* que vai variar sobre uma sequência de 1 a 10. Perceba abaixo, que o `for()` *loop* não está executando nada além de um simples comando de `print()` sobre o valor assumido pelo índice *i*. Ou seja, a cada repetição, tudo o que `for()` *loop* está fazendo é mostrar qual o valor que o índice *i* assumiu nessa repetição.

```
for(i in 1:10){  
    print(i)  
}
```

¹Ou seja, não estou me referindo as letras 'i's que aparecem no meio de palavras como `print`.

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

Portanto, no exemplo acima, ao estipularmos na definição de *loop*, o conjunto de valores sobre os quais o índice do *loop* iria variar, incluímos uma sequência de 1 a 10, através do código 1:10. Entretanto, nós podemos incluir o que quisermos como conjunto de valores, desde que eles estejam em algum vetor (seja esse vetor, um vetor atômico ou uma lista). Por exemplo, nós podemos colocar um vetor de nomes, como no exemplo abaixo.

```
vec <- c("Ana", "Eduardo", "Heloísa")

for(i in vec){
  print(i)
}

## [1] "Ana"
## [1] "Eduardo"
## [1] "Heloísa"
```

Porém, uma forma mais interessante e poderosa de se utilizar o índice de seu *loop*, é através de um mecanismo de *subsetting*. Ou seja, você pode utilizar o índice de seu *loop*, como um índice de *subsetting*, em cada repetição do *loop*. Dessa forma, a cada repetição, você aplica as funções definidas no corpo do *loop*, em uma parte diferente de seus dados. O índice de seu *loop*, passa a ser o índice que representa uma parte específica de seus objetos, ao ser utilizado pelas funções de *subsetting* (funções `[]` e `[[`). Lembre-se que podemos utilizar diferentes tipos de índice dentro das funções de *subsetting*, como demonstrado na seção [Subsetting](#), porém, no caso de um *loop*, os índices de tipo numérico (integer) e textual (character) são os mais úteis.

Por exemplo, logo abaixo, temos uma lista que contém alguns valores numéricos. Porém, eu dei um nome a cada um desses valores numéricos. Perceba abaixo, que no corpo do *loop* (mais especificamente no código `lista[[i]]`), estamos utilizando a função `[[` para extrair o elemento do objeto `lista`, que possui o nome igual ao valor que o nosso índice `i` assume em uma dada repetição. Dessa forma, nós estamos multiplicando por 10, apenas os elementos de `lista`, que estão nomeados de acordo com os valores contidos no vetor `vec`, que representa o conjunto de valores que o índice do *loop* vai assumir. Ou seja, nós podemos utilizar um `for()` *loop*, para aplicarmos uma função apenas sobre algumas partes específicas de nossos dados (deixando outras partes intactas). Perceba abaixo,

que em nenhum momento o *loop* chega a lidar com o valor 40, que está nomeado como Marcos no objeto `lista`.

```
vec <- c("Ana", "Eduardo", "Heloísa")
```

```
lista <- list(  
  "Ana" = 15,  
  "Eduardo" = 30,  
  "Heloísa" = 10,  
  "Marcos" = 40  
)
```

```
for(i in vec){  
  print(lista[[i]])  
  print(lista[[i]] * 10)  
}
```

```
## [1] 15  
## [1] 150  
## [1] 30  
## [1] 300  
## [1] 10  
## [1] 100
```

5.5.1 O problema do vetor crescente

Para salvar os resultados gerados em cada repetição do `for()` *loop*, é **muito importante** que você crie previamente, um objeto que possa guardar esses resultados (GILLESPIE; LOVELACE, 2017; WICKHAM; GROLEMUND, 2017). Em outras palavras, você precisa reservar o espaço necessário para guardar os resultados, antes mesmo de executar o `for()` *loop*. Por exemplo, se eu possuo um `data.frame` contendo 4 colunas numéricas, e desejo calcular a média de cada coluna, eu preciso criar algum objeto que possa receber as 4 médias que serão geradas pelo *loop*. No exemplo abaixo, eu utilizo a função `vector()` para criar um novo vetor atômico chamado `media`, que é do tipo `double` e que possui 4 elementos. Logo, dentro do corpo do *loop*, eu salvo os resultados da função `mean()` dentro de cada elemento desse vetor `media`.

```
df <- data.frame(  
  a = rnorm(20),  
  b = rnorm(20),  
  c = rnorm(20),  
  d = rnorm(20)  
)
```

```
media <- vector(mode = "double", length = 4)  
for(i in 1:4){
```

```
media[i] <- mean(df[[i]])
}

media

## [1] 0.31714622 -0.02185939 -0.14591883 0.17135666
```

Esse problema é conhecido por muitos usuários, como o *growing vector problem* (ou “problema do vetor crescente”). Caso você se esqueça de reservar previamente o espaço para guardar os resultados, o seu *loop* será bem lento. Pois a cada repetição do *loop*, um novo resultado é gerado e, portanto, o computador tem que reservar um tempo para aumentar o vetor onde esse resultado será guardado, de forma que aquele novo resultado “caiba” neste vetor. Ou seja, se o meu *loop* vai gerar 2 mil repetições, o meu computador vai ter que parar 2 mil vezes durante o processo, para aumentar um elemento a mais em meu vetor, com o objetivo de guardar o novo resultado gerado em cada uma dessas 2 mil repetições. Por outro lado, se eu já reservo previamente um vetor com 2 mil elementos, o meu computador não precisa parar durante o processo, pois todo o espaço necessário já está preparado e a espera do novo resultado gerado em cada repetição.

Por isso, você precisa sempre pensar sobre quais tipos de resultados serão gerados em seu *loop*. Ou seja, será que a cada repetição de seu *loop*, um único número é gerado (por exemplo, uma média)? Ou um vetor (podemos aplicar um teste lógico em cada coluna, e ter um vetor lógico como resultado)? Ou uma tabela (na próxima seção, damos um exemplo desse caso)? A partir do momento que você sabe qual o tipo de resultado que será gerado pelo seu *loop*, você pode identificar com mais facilidade, qual a melhor estrutura para guardar esses valores. Pergunte-se: será que um vetor atômico consegue guardar esses resultados? Ou uma lista é mais adequada? Lembre-se que vetores atômicos só podem guardar dentro de si, valores que pertencem ao mesmo tipo de dado (*double*, *integer*, *logical*, *character*, etc.). Além disso, nós não podemos guardar um vetor, ou uma tabela, ou em outras palavras, um conjunto de valores dentro de cada elemento de um vetor atômico. Logo, caso o resultado de cada repetição de seu *loop* seja, por exemplo, uma tabela, é melhor que você utilize uma lista para guardá-las.

5.6 Um estudo de caso: uma demanda real sobre a distribuição de ICMS

Nessa seção, vou apresentar um exemplo prático, sobre uma demanda real que chegou até mim em 2020. Na época, eu trabalhava como estagiário na Diretoria de Estatística e Informações da Fundação João Pinheiro (FJP-MG), mais especificamente com uma lei estadual que é tradicionalmente chamada de Lei Robin Hood (Lei 18.030 de 2009 - MG). Essa lei rege a distribuição do ICMS total de Minas Gerais, ao longo dos municípios do estado.

Em resumo, o Governo de Minas Gerais, coleta o ICMS (imposto sobre operações relativas à circulação de mercadorias e sobre prestações de serviços de transporte interestadual, intermunicipal e de comunicação) gerado em todo o estado, e ao final de um período, ele redistribui esse valor para

os 853 municípios do estado. Cada município, possui um índice de participação, que corresponde à porcentagem do ICMS total ao qual o respectivo município tem direito. Em outras palavras, se o ICMS total gerado no estado em um período foi de 8,5 bilhões de reais, e o município de Belo Horizonte possui um índice de participação equivalente a 0,009, isso significa que ao final do período, 0,9% do ICMS total, ou 76,5 milhões de reais serão transferidos para a prefeitura do município de Belo Horizonte.

A lei possui diversos critérios presentes no cálculo do índice de participação de cada município, sendo alguns deles: Turismo, Esporte, Patrimônio Cultural, População e Receita Própria. Em suma, o índice de participação de cada município, é calculado a partir de uma média ponderada dos índices de cada um desses diversos critérios da lei. Você pode encontrar uma descrição completa desses critérios e do cálculo dos índices de participação, no texto original da lei².

5.6.1 A demanda em si

A demanda é muito simples, porém, ela é trabalhosa e envolve um volume excessivo de repetição se você optar por utilizar programas como Excel para resolvê-la. Dentre os vários critérios da lei, temos o critério de Meio Ambiente, e o órgão responsável pelo cálculo do índice referente a esse critério, é a SEMAD-MG (Secretaria de Estado de Meio Ambiente e Desenvolvimento Sustentável). Um dia, a SEMAD chegou até nós da Fundação João Pinheiro (FJP), pedindo por todos os valores de ICMS transferidos para cada município, ao longo dos anos de 2018 e 2019, de acordo com o critério do Meio Ambiente da Lei Robin Hood.

Nós da FJP, calculamos todo mês, os valores transferidos de ICMS separados por cada critério da lei, e para cada município. Ou seja, para o ano de 2019, pense por exemplo, em uma lista de arquivos de Excel parecida com a lista abaixo, onde cada planilha corresponde aos valores de ICMS transferidos em um mês específico do ano.

Dando uma olhada mais de perto, cada uma dessas planilhas do Excel, assumem a estrutura abaixo. Onde cada linha da tabela, representa um município do estado de Minas Gerais, e cada coluna (ou pelo menos, grande parte dessas colunas), representa os valores de ICMS transferidos segundo os índices de um critério específico da lei. Ou seja, a coluna Educação, nos apresenta os valores de ICMS transferidos para cada município do estado, considerando-se o índice que cada um desses municípios adquiriram no critério de Educação, e também, considerando-se a parcela que o critério de Educação representa do total de ICMS distribuído.

```
library(readxl)
```

```
Abril_2019 <- read_excel("planilhas/Abril_2019.xlsx")
```

```
Abril_2019
```

²<<https://www.almg.gov.br/consulte/legislacao/completa/completa-nova-min.html?tipo=LEI&num=18030&comp=&ano=2009&texto=original>>

Figura 5.5: Lista de arquivos do Excel

Nome	Data de modificação	Tipo	Tamanho
 Abril_2019	18/08/2020 16:51	Planilha do Micro...	234 KB
 Agosto_2019	18/08/2020 17:13	Planilha do Micro...	235 KB
 Dezembro_2019	18/08/2020 16:53	Planilha do Micro...	232 KB
 Fevereiro_2019	18/08/2020 16:51	Planilha do Micro...	252 KB
 Janeiro_2019	18/08/2020 16:49	Planilha do Micro...	254 KB
 Julho_2019	18/08/2020 16:54	Planilha do Micro...	233 KB
 Junho_2019	18/08/2020 16:55	Planilha do Micro...	233 KB
 Maio_2019	18/08/2020 16:56	Planilha do Micro...	233 KB
 Março_2019	18/08/2020 16:59	Planilha do Micro...	250 KB
 Novembro_2019	18/08/2020 17:00	Planilha do Micro...	251 KB
 Outubro_2019	18/08/2020 17:02	Planilha do Micro...	233 KB
 Setembro_2019	18/08/2020 17:02	Planilha do Micro...	233 KB

Fonte: Elaboração própria do autor.

```
## # A tibble: 853 x 27
##   IBGE1 IBGE2 SEF Municípios População `População dos 50 + Popu~
##   <dbl> <dbl> <dbl> <chr>          <dbl>          <dbl>
## 1 310010 10 1 ABADIA DOS DOURA~ 8847.          0
## 2 310020 20 2 ABAETÉ          29470.         0
## 3 310030 30 3 ABRE CAMPO      17087.         0
## 4 310040 40 4 ACAIACA         5068.         0
## 5 310050 50 5 AÇUCENA         12151.         0
## 6 310060 60 6 ÁGUA BOA        17258.         0
## 7 310070 70 7 ÁGUA COMPRIDA    2544.          0
## 8 310080 80 8 AGUANIL          5644.          0
## 9 310090 90 9 ÁGUAS FORMOSAS   24321.         0
## 10 310100 100 10 ÁGUAS VERMELHAS 17102.         0
## # ... with 843 more rows, and 21 more variables: Área Geográfica <dbl>,
## # Educação <dbl>, Patrimônio Cultural <dbl>, Receita Própria <dbl>,
## # Cota Mínima <dbl>, Mineradores <dbl>, Saúde Per Capita <dbl>,
## # VAF <dbl>, Esportes <dbl>, Turismo <dbl>, Penitenciárias <dbl>,
## # Recursos Hídricos <dbl>, Produção de Alimentos 2º semestre <dbl>,
## # Unidades de Conservação (IC i) <dbl>, Saneamento <dbl>,
## # Mata Seca <dbl>, Meio Ambiente <dbl>, PSF <dbl>, ICMS Solidário <dbl>,
## # Índice Mínimo per capita <dbl>, Total <dbl>
```

Porém, temos dois problemas aqui: 1) A SEMAD precisa apenas dos valores de ICMS transferidos de acordo com o critério de Meio Ambiente, e nada mais; 2) A SEMAD precisa dos valores de ICMS transferidos ao longo de todos os meses dos anos de 2018 e 2019, e se nós temos 12 planilhas por ano, temos que reunir então 24 planilhas para a secretaria. Alguns poderiam argumentar que

esses pontos não se configuram como problemas de fato. Em outras palavras, algumas pessoas poderiam dizer algo como: “Ora, você precisa apenas pegar as 24 planilhas, depois juntá-las em um arquivo .zip, e simplesmente enviar esse arquivo para a SEMAD. A SEMAD que se vire para coletar e organizar as informações dessas planilhas!”.

Entretanto, essa não é uma boa política de trabalho. Transportando esse problema para um ambiente mais corporativo, o excesso de informação traz dificuldades para a tomada de decisão da empresa, pois ele camufla aquelas pequenas informações que são de fato relevantes. Ou seja, carregar apenas as informações que são relevantes para a empresa, representa uma boa prática. Além disso, o pedido da SEMAD é simples, e os problemas pontuados acima são de rápida e fácil solução no R, através do uso de funções e *loop*'s.

Como exemplo prático, vamos mostrar a metodologia apenas para essas 12 planilhas referentes ao ano de 2019, e você pode replicar facilmente essa metodologia para as outras 12, 24, 36 ou quantas outras planilhas desejar. Em suma, a metodologia que vou apresentar envolve duas etapas: 1) juntar todas essas 12 planilhas, em uma planilha só. Ou seja, desejamos guardar em uma planilha única, os valores transferidos aos municípios de MG em cada mês do ano de 2019; 2) Selecionar apenas as colunas relevantes para a SEMAD no arquivo final. Tendo essas etapas em mente, isso seria um trabalho muito repetitivo no Excel. Uma repetição que é cansativa, e que pode facilmente te levar a erros no processo. Aqueles com mais experiência no Excel, poderiam utilizar a plataforma de *queries* do programa para carregar os dados de cada planilha. Porém, apenas pelo tempo que você levaria para configurar a sua *querie*, seria muito mais rápido se você simplesmente adotasse a estratégia de `Crtl+C` e `Crtl+V`, para transferir os dados em uma planilha única.

Por outro lado, traduzindo essa metodologia para o R, precisaríamos realizar as seguintes etapas: 1) construir uma função, que será responsável por importar essas planilhas para o R, além de já inserir uma coluna que define o mês a que os dados se referem, e de selecionar apenas as colunas relevantes para a SEMAD; 2) em seguida, podemos utilizar um *loop* para replicar a função criada ao longo de todas as planilhas disponíveis na pasta; 3) por fim, uní-las em uma única tabela com a função `bind_rows()`. Ou seja, o objetivo é ler todas as 12 planilhas com apenas um comando, e em seguida, utilizar um segundo comando para unir essas planilhas em uma só.

Tendo isso em mente, o primeiro passo é criarmos uma função que será responsável por importar e configurar os dados de cada planilha. Antes disso, vamos coletar os nomes de cada planilha, através da função `list.files()`. Como o próprio nome dá a entender, essa função busca listar os nomes de todos os arquivos contidos em uma pasta. Caso você não defina alguma pasta específica na função (diferente do que fizemos abaixo), `list.files()` vai listar todos os arquivos presentes no seu diretório de trabalho atual do R.

```
nomes_planilhas <- list.files("planilhas/")
```

```
nomes_planilhas
```

```
## [1] "Abril_2019.xlsx"      "Agosto_2019.xlsx"    "Dezembro_2019.xlsx"
## [4] "Fevereiro_2019.xlsx" "Janeiro_2019.xlsx"   "Julho_2019.xlsx"
```

```
## [7] "Junho_2019.xlsx"      "Maio_2019.xlsx"       "Marco_2019.xlsx"
## [10] "Novembro_2019.xlsx"   "Outubro_2019.xlsx"    "Setembro_2019.xlsx"
```

Em resumo, a nossa função (chamada `ler_excel()`) vai receber como *input*, o nome de cada planilha que acabamos de coletar através da função `list.files()`, e vai aplicar 3 etapas principais: 1) importar para o R, a planilha que contém o nome que fornecemos como *input*; 2) a função vai selecionar apenas as colunas relevantes à SEMAD (as colunas que contém os códigos de cada município, e a coluna Meio Ambiente); 3) em seguida, essa função vai criar três novas colunas nessa planilha, que contém o nome da planilha de onde esses dados vieram, além do mês e do ano, que estão implícitos no nome dessa planilha.

Para importarmos as planilhas do Excel, podemos utilizar a função `read_excel()`, que introduzimos na seção [Importando arquivos em Excel com readxl](#). Porém, para fornecermos os endereços corretos de cada planilha à função `read_excel()`, nós precisamos criar um mecanismo que possa colar o caminho até a pasta onde se encontram esses arquivos, ao nome de cada planilha. Algo que podemos facilmente resolver com o uso da função `paste()`. Com essa função, podemos unir o nome da pasta onde as planilhas se encontram (pasta planilhas), ao nome de cada planilha que a função receber como *input*.

Para adicionarmos novas colunas à planilha, nós podemos utilizar a função `mutate()`, que introduzimos na seção [Adicionando variáveis à nossa tabela com mutate\(\)](#). Essas novas colunas são necessárias, pois vão conter informações essenciais como o mês e ano a que os dados se referem. Dentre essas novas colunas, teremos a coluna *Origem*, que vai conter apenas o nome da planilha que está sendo lida pela função, e que portanto, guarda a origem dos dados selecionados pela função. Dessa forma, nós teremos dentro dos nossos dados, uma coluna que é capaz de nos informar a planilha de onde os dados vieram.

Em contrapartida, como o mês e o ano a que os dados de cada planilha se referem, estão implícitos no próprio nome da planilha, precisamos de funções que possam lidar com *input*'s textuais. Para extrairmos os anos de cada planilha, nós podemos utilizar a função `parse_number()` (que é capaz de extrair números que se encontram em uma cadeia de texto), guardando o resultado dessa função, em uma coluna chamada *Ano*. Entretanto, para extrairmos o mês do nome de cada planilha, teremos um pouco mais de trabalho. Pois os nomes de cada planilha variam em comprimento, e por isso, não posso simplesmente extrair os 5, 6, ou 7 primeiros caracteres de cada nome. A melhor alternativa, é utilizarmos a função `str_length()` para calcularmos o número total de caracteres em cada nome, e em seguida, eliminarmos a parte que permanece constante em todos os nomes das planilhas (mais especificamente, a parte `_2019`).

```
library(readxl)
library(tidyverse)

ler_excel <- function(x){

  caminho <- paste("planilhas/", x, sep = "")
```

```

planilha <- read_excel(caminho) %>%
  mutate(
    Origem = x,
    Ano = parse_number(x),
    Mês = str_sub(x, start = 1, end = str_length(x) - 10)
  ) %>%
  select(IBGE1, `Municípios`, Origem, Ano, `Mês`, `Meio Ambiente`)

return(planilha)
}

```

Com a nossa função `ler_excel()` construída, podemos aplicá-la sobre o primeiro nome contido em nosso objeto `nomes_planilhas`, como um teste. Pelo resultado abaixo, nós podemos confirmar que a função está funcionando exatamente como esperávamos.

```

ler_excel(nomes_planilhas[1])

## # A tibble: 853 x 6
##   IBGE1 Municípios      Origem      Ano Mês `Meio Ambiente`
##   <dbl> <chr>          <chr>    <dbl> <chr>      <dbl>
## 1 310010 ABADIA DOS DOURADOS Abril_2019.xlsx 2019 Abril          0
## 2 310020 ABAETÉ          Abril_2019.xlsx 2019 Abril          0
## 3 310030 ABRE CAMPO      Abril_2019.xlsx 2019 Abril      10433.
## 4 310040 ACAIACA         Abril_2019.xlsx 2019 Abril          0
## 5 310050 AÇUCENA         Abril_2019.xlsx 2019 Abril      7727.
## 6 310060 ÁGUA BOA        Abril_2019.xlsx 2019 Abril      10433.
## 7 310070 ÁGUA COMPRIDA   Abril_2019.xlsx 2019 Abril      21909.
## 8 310080 AGUANIL         Abril_2019.xlsx 2019 Abril          0
## 9 310090 ÁGUAS FORMOSAS  Abril_2019.xlsx 2019 Abril          0
## 10 310100 ÁGUAS VERMELHAS Abril_2019.xlsx 2019 Abril      14750.
## # ... with 843 more rows

```

Agora que temos a função construída, nós somos capazes de utilizar um `for()` *loop* para replicar a função `ler_excel()` sobre cada uma das 12 planilhas presentes na pasta `planilhas` do meu computador. Porém, é muito importante destacar, que toda vez em que você tiver que guardar os resultados de seu `for()` *loop*, você **sempre** deve criar antes de executar o *loop*, alguma estrutura que seja capaz de guardar os resultados deste *loop*. Caso você não realize esse processo, é muito possível que o seu *loop* se torne muito lento, pelo fato de que o computador terá de guardar um certo tempo (durante a execução do *loop*), para se preocupar em alocar esses resultados na memória de seu computador.

Como definimos na seção **O problema do vetor crescente**, você precisa pensar sobre qual a estrutura de dado é a mais adequada para receber (ou armazenar) os resultados de seu *loop*. No exemplo dessa seção, o nosso *loop* está gerando a cada repetição, uma tabela, ou um `data.frame`. Nós sabemos que serão 12 `data.frame`'s gerados, e portanto, precisamos de uma estrutura que seja capaz de guardar essas 12 tabelas. Para o nosso caso, uma lista é a melhor opção, pois podemos incluir o

que quisermos em cada elemento de uma lista. Por isso eu crio uma lista chamada `planilhas`, antes de executar o `loop`. Essa lista possui 12 elementos (que no momento estão vazios), onde serão guardados cada uma das tabelas resultantes da nossa função `ler_excel()`.

Perceba também abaixo, que dentro da definição do `loop`, eu utilizo a função `seq_along()` sobre o objeto `nomes_planilhas`. Essa função é um atalho útil, para criarmos uma sequência que vai de 1 até o número total de elementos presentes no objeto `nomes_planilhas`. Ou seja, a função `seq_along()` gera exatamente o mesmo resultado do que o comando `1:length(nomes_planilhas)`. Porém, a função `seq_along(x)` (ao contrário do comando `1:length(x)`) se comporta de maneira adequada em situações extremas³, e por isso, representa uma forma mais segura de criarmos uma sequência numérica que irá servir como o conjunto de valores sobre o qual o índice do `loop` vai atuar.

```
planilhas <- vector(mode = "list", length = length(nomes_planilhas))

for(i in seq_along(nomes_planilhas)){

  planilhas[[i]] <- ler_excel(nomes_planilhas[i])

}
```

Após executarmos o `loop`, cada uma das 12 planilhas foram guardadas em cada elemento da lista `planilhas`. Perceba abaixo, que cada uma das planilhas se encontram agora no formato que esperávamos, após aplicarmos a função `ler_excel()`, pois temos apenas as colunas que a SEMAD necessita.

```
planilhas[[3]]
```

## # A tibble: 853 x 6					
##	IBGE1 Municípios	Origem	Ano Mês		`Meio Ambiente`
##	<dbl> <chr>	<chr>	<dbl> <chr>		<dbl>
##	1 310010 ABADIA DOS DOURAD~	Dezembro_2019.x~	2019 Dezemb~		0
##	2 310020 ABAETÉ	Dezembro_2019.x~	2019 Dezemb~		0
##	3 310030 ABRE CAMPO	Dezembro_2019.x~	2019 Dezemb~		0
##	4 310040 ACAIACA	Dezembro_2019.x~	2019 Dezemb~		0
##	5 310050 AÇUCENA	Dezembro_2019.x~	2019 Dezemb~		2617.
##	6 310060 ÁGUA BOA	Dezembro_2019.x~	2019 Dezemb~		0
##	7 310070 ÁGUA COMPRIDA	Dezembro_2019.x~	2019 Dezemb~		0
##	8 310080 AGUANIL	Dezembro_2019.x~	2019 Dezemb~		16776.
##	9 310090 ÁGUAS FORMOSAS	Dezembro_2019.x~	2019 Dezemb~		0

³O principal exemplo, é quando o seu objeto possui comprimento 0, ou 0 elementos. Essa situação geralmente ocorre de maneira não proposital, sendo resultado de algum erro durante alguma etapa de seus cálculos. Em um caso como esse, o comando `1:length(x)` equivale ao comando `1:0`, e como resultado, a sequência `c(1, 0)` será gerada. Já a função `seq_along(x)`, vai gerar um vetor numérico vazio nesse caso. Um vetor numérico vazio evita que o nosso `loop` seja executado, e essa é a ação mais apropriada para um exemplo como esse.

```
## 10 310100 ÁGUAS VERMELHAS      Dezembro_2019.x~  2019 Dezemb~      21705.
## # ... with 843 more rows
```

Porém, ainda não atingimos o resultado desejado. Lembre-se que cada uma das 12 planilhas, se encontram no momento, separadas em cada elemento de uma lista. Nós estabelecemos anteriormente, que o ideal seria reunirmos todas essas 12 tabelas, em uma só. Para executarmos esse passo, nós podemos simplesmente aplicar a função `bind_rows()` sobre a lista `planilhas`. Como o próprio nome da função dá a entender, ela busca unir, ou colar linhas de diferentes tabelas. Em uma outra perspectiva, é como se a função `bind_rows()` estivesse “empilhando” as tabelas, uma em cima da outra. Portanto, se nós temos 12 tabelas diferentes, onde cada linha de cada uma dessas tabelas representa um dos 853 municípios de Minas Gerais, ao unirmos todas essas tabelas, deveríamos ter como resultado, uma única tabela contendo 10.236 linhas ($853 \times 12 = 10.236$).

```
planilhas <- bind_rows(planilhas)
```

```
planilhas
```

```
## # A tibble: 10,236 x 6
##   IBGE1 Municípios      Origem      Ano Mês  `Meio Ambiente`
##   <dbl> <chr>          <chr>      <dbl> <chr>          <dbl>
## 1 310010 ABADIA DOS DOURADOS Abril_2019.xlsx 2019 Abril            0
## 2 310020 ABAETÉ          Abril_2019.xlsx 2019 Abril            0
## 3 310030 ABRE CAMPO      Abril_2019.xlsx 2019 Abril      10433.
## 4 310040 ACAIACA         Abril_2019.xlsx 2019 Abril            0
## 5 310050 AÇUCENA         Abril_2019.xlsx 2019 Abril      7727.
## 6 310060 ÁGUA BOA        Abril_2019.xlsx 2019 Abril      10433.
## 7 310070 ÁGUA COMPRIDA   Abril_2019.xlsx 2019 Abril      21909.
## 8 310080 AGUANIL         Abril_2019.xlsx 2019 Abril            0
## 9 310090 ÁGUAS FORMOSAS   Abril_2019.xlsx 2019 Abril            0
## 10 310100 ÁGUAS VERMELHAS Abril_2019.xlsx 2019 Abril      14750.
## # ... with 10,226 more rows
```

5.6.2 Conclusão

Uma tarefa que inicialmente seria trabalhosa e extremamente repetitiva em muitos programas comuns (como o Excel), pode ser resolvida no R de maneira fácil e rápida, através do uso de uma função e um *loop*. Tínhamos como objetivo, reunir os dados presentes em 12 planilhas em uma única tabela, e em seguida, selecionar apenas aquelas colunas que eram de interesse da SEMAD. Se reunirmos todos os comandos que utilizamos no R, temos um *script* com mais ou menos 30 linhas. Ou seja, com apenas 30 linhas, nós economizamos um tempo e esforço enormes em nosso trabalho.

```
library(readxl)
library(tidyverse)
```

```
nomes_planilhas <- list.files("planilhas/")

ler_excel <- function(x){

  caminho <- paste("planilhas/", x, sep = "")

  planilha <- read_excel(caminho) %>%
    mutate(
      Origem = x,
      Ano = parse_number(x),
      Mês = str_sub(x, start = 1, end = str_length(x) - 10)
    ) %>%
    select(IBGE1, `Municípios`, Origem, Ano, `Mês`, `Meio Ambiente`)

  return(planilha)
}

planilhas <- vector(mode = "list", length = length(nomes_planilhas))

for(i in seq_along(nomes_planilhas)){

  planilhas[[i]] <- ler_excel(nomes_planilhas[i])

}

planilhas <- bind_rows(planilhas)
```

A partir daqui, com a tabela única em nossas mãos, nós precisamos apenas exportar essa tabela para fora do R. Algo que pode ser rapidamente realizado através de uma função como a `write_csv2()`, que introduzimos na seção [Exportando dados em arquivos de texto com readr](#).

Capítulo 6

Introdução a base de dados relacionais no R

6.1 Introdução e pré-requisitos

Segundo [Nield \(2016, p.53\)](#), *joins* são uma das funcionalidades que definem a linguagem SQL (*Structured Query Language*). Por isso, *joins* são um tipo de operação muito relacionado à RDBMS (*Relational DataBase Management Systems*), que em sua maioria, utilizam a linguagem SQL. Logo, essa seção será muito familiar para aqueles que possuem experiência com essa linguagem.

Para executarmos uma operação de *join*, os pacotes básicos do R oferecem a função `merge()`. Entretanto, vamos abordar o pacote `dplyr` neste capítulo, que também possui funções especializadas neste tipo de operação. Com isso, para ter acesso às funções que vamos mostrar aqui, você pode chamar tanto pelo pacote `dplyr` quanto pelo `tidyverse`.

```
library(tidyverse)
library(dplyr)
```

6.2 Dados relacionais e o conceito de *key*

Normalmente, trabalhamos com diversas bases de dados diferentes ao mesmo tempo. Pois é muito incomum, que uma única tabela contenha todas as informações das quais necessitamos e, por isso, transportar os dados de uma tabela para outra se torna uma atividade essencial em muitas ocasiões.

Logo, de alguma maneira, os dados presentes nessas diversas tabelas se relacionam entre si. Por exemplo, suponha que você possua uma tabela contendo o PIB dos municípios do estado de Minas Gerais, e uma outra tabela contendo dados demográficos desses mesmos municípios. Se você deseja unir essas duas tabelas em uma só, você precisa de algum mecanismo que possa conectar um valor do município X na tabela A com a linha da tabela B correspondente ao mesmo município X, e através dessa conexão, conduzir o valor da tabela A para esse local específico da tabela B, ou vice-versa. O processo que realiza esse cruzamento entre as informações, e que por fim, mescla ou funde as duas tabelas de acordo com essas conexões, é chamado de *join*.

Por isso, dizemos que os nossos dados são “relacionais”. Pelo fato de que nós possuímos diversas tabelas que descrevem os mesmos indivíduos, municípios, firmas ou eventos. Mesmo que essas tabelas estejam trazendo variáveis ou informações muito diferentes desses indivíduos, elas possuem essa característica em comum e, com isso, possuem uma relação entre si, e vamos frequentemente nos aproveitar dessa relação para executarmos análises mais completas.

Porém, para transportarmos esses dados de uma tabela a outra, precisamos de alguma chave, ou de algum mecanismo que seja capaz de identificar as relações entre as duas tabelas. Em outras palavras, se temos na tabela A, um valor pertencente ao indivíduo X, e queremos transportar esse valor para a tabela B, nós precisamos de algum meio que possa identificar o local da tabela B que seja referente ao indivíduo X. O mecanismo que permite essa comparação, é o que chamamos de *key* ou de “chave”.

```
d <- c("1943-07-26", "1940-09-10", "1942-06-18", "1943-02-25", "1940-07-07")
```

```
info <- tibble(
  name = c("Mick", "John", "Paul", "George", "Ringo"),
  band = c("Rolling Stones", "Beatles", "Beatles", "Beatles", "Beatles"),
  born = as.Date(d),
  children = c(TRUE)
)

band_instruments <- tibble(
  name = c("John", "Paul", "Keith"),
  plays = c("guitar", "bass", "guitar")
)
```

Como exemplo inicial, vamos utilizar a tabela `info`, que descreve características pessoais de um conjunto de músicos famosos. Também temos a tabela `band_instruments`, que apenas indica qual o instrumento musical utilizado por parte dos músicos descritos na tabela `info`.

```
info
```

```
## # A tibble: 5 x 4
##   name   band      born      children
##   <chr> <chr>    <date>    <lgl>
## 1 Mick   Rolling Stones 1943-07-26 TRUE
## 2 John   Beatles      1940-09-10 TRUE
## 3 Paul   Beatles      1942-06-18 TRUE
## 4 George Beatles      1943-02-25 TRUE
## 5 Ringo Beatles      1940-07-07 TRUE
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul   bass
## 3 Keith guitar
```

Portanto, precisamos de uma *key* para detectarmos as relações entre as tabelas `info` e `band_instruments`. Uma *key* consiste em uma variável (ou um conjunto de variáveis), que é capaz de identificar unicamente cada indivíduo descrito em uma tabela, sendo que essa variável (ou esse conjunto de variáveis), deve obrigatoriamente estar presente em ambas as tabelas em que desejamos aplicar o *join*. Dessa forma, podemos através dessa variável, discernir quais indivíduos estão presentes nas duas tabelas, e quais se encontram em apenas uma delas.

Ao observar as tabelas `info` e `band_instruments`, você talvez perceba que ambas possuem uma coluna denominada `name`. No nosso caso, essa é a coluna que representa a *key* entre as tabelas `info` e `band_instruments`. Logo, ao identificar o músico que está sendo tratado em cada linha, a coluna `name` nos permite cruzar as informações existentes em ambas tabelas. Com isso, podemos observar que os músicos John e Paul, estão disponíveis em ambas as tabelas, mas os músicos Mick, George e Ringo estão descritos apenas na tabela `info`, enquanto o músico Keith se encontra apenas na tabela `band_instruments`.

Segundo [Nield \(2016\)](#), podemos ter dois tipos de *keys* existentes em uma tabela:

- 1) *Primary key*: uma variável capaz de identificar unicamente cada uma das observações presentes em sua tabela.
- 2) *Foreign key*: uma variável capaz de identificar unicamente cada uma das observações presentes em uma outra tabela.

Com essas características em mente, podemos afirmar que a coluna `name` existente nas tabelas `info` e `band_instruments`, se trata de uma *primary key*. Pois em ambas as tabelas, mais especificamente em cada linha dessa coluna, temos um músico diferente, ou em outras palavras, não há um músico duplicado.

Por outro lado, uma *foreign key* normalmente contém valores repetidos ao longo da base e, por essa razão, não são capazes de identificar unicamente uma observação na tabela em que se encontram. Porém, os valores de uma *foreign key* certamente fazem referência a uma *primary key* existente em uma outra tabela. Tendo isso em mente, o objetivo de uma *foreign key* não é o de identificar cada observação presente em uma tabela, mas sim, de indicar ou explicitar a relação que a sua tabela possui com a *primary key* presente em uma outra tabela.

Por exemplo, suponha que eu tenha a tabela `children` abaixo. Essa tabela descreve os filhos de alguns músicos famosos, e a coluna `father` caracteriza-se como a *foreign key* dessa tabela. Não apenas porque os valores da coluna `father` se repetem ao longo da base, mas também, porque essa coluna pode ser claramente cruzada com a coluna `name` pertencente às tabelas `info` e `band_instruments`.

Figura 6.1: Cruzamento entre as tabelas de acordo com a coluna name

Músico presente apenas na tabela info

Músico presente em ambas as tabelas

Músico presente apenas na tabela band_instruments

Tabela info

name	band	born	children
Mick	Rolling Stones	1943-07-26	TRUE
John	Beatles	1940-09-10	TRUE
Paul	Beatles	1942-06-18	TRUE
George	Beatles	1943-02-25	TRUE
Ringo	Beatles	1940-07-07	TRUE

Tabela band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar

Fonte: Elaboração própria do autor.

```
children <- tibble(
  child = c("Stella", "Beatrice", "James", "Mary",
            "Heather", "Sean", "Julian", "Zak",
            "Lee", "Jason", "Dhani"),
  sex = c("F", "F", "M", "F", "F", "M", "M", "M", "F", "M", "M"),
  father = c(rep("Paul", times = 5), "John", "John",
             rep("Ringo", times = 3), "Harrison")
)
```

```
children
```

```
## # A tibble: 11 x 3
##   child    sex father
##   <chr>   <chr> <chr>
## 1 Stella  F     Paul
## 2 Beatrice F     Paul
## 3 James   M     Paul
## 4 Mary    F     Paul
## 5 Heather F     Paul
## 6 Sean    M     John
## 7 Julian  M     John
## 8 Zak     M     Ringo
## 9 Lee     F     Ringo
## 10 Jason  M     Ringo
## 11 Dhani  M     Harrison
```

6.3 Introduzindo *joins*

Tendo esses pontos em mente, o pacote `dplyr` nos oferece quatro funções voltadas para operações de *join*. Cada uma dessas funções executam um tipo de *join* diferente, que vamos comentar na próxima seção. Por agora, vamos focar apenas na função `inner_join()`, que como o seu próprio nome dá a entender, busca aplicar um *inner join*.

Para utilizar essa função, precisamos nos preocupar com três argumentos principais. Os dois primeiros argumentos (`x` e `y`), definem os `data.frame`'s a serem fundidos pela função. Já no terceiro argumento (`by`), você deve delimitar a coluna, ou o conjunto de colunas que representam a *key* entre as tabelas fornecidas em `x` e `y`.

Assim como em qualquer outro tipo de *join*, as duas tabelas envolvidas serão unidas, porém, em um *inner join*, apenas as linhas de indivíduos que se encontram em ambas as tabelas serão retornadas na nova tabela gerada. Perceba abaixo, que a função `inner_join()` criou uma nova tabela contendo todas as colunas presentes nas tabelas `info` e `band_instruments` como esperávamos, e que ela manteve apenas as linhas referentes aos músicos John e Paul, que são os únicos indivíduos que aparecem em ambas as tabelas.

```
inner_join(info, band_instruments, by = "name")
```

```
## # A tibble: 2 x 5
##   name band   born      children plays
##   <chr> <chr>   <date>    <lgl>    <chr>
## 1 John Beatles 1940-09-10 TRUE     guitar
## 2 Paul Beatles 1942-06-18 TRUE     bass
```

```
## -----
```

```
## A mesma operação com o uso do pipe ( %>% ):
```

```
info %>%
  inner_join(band_instruments, by = "name")
```

Ao observar esse resultado, você talvez chegue a conclusão de que um processo de *join* se trata do mesmo processo executado pela função `PROCV()` do Excel. Essa é uma ótima comparação! Pois a função `PROCV()` realiza justamente um *join* parcial, ao trazer para a tabela A, uma coluna pertencente a tabela B, de acordo com uma *key* que conecta as duas tabelas.

Por outro lado, nós não podemos afirmar que a função `PROCV()` busca construir um *join per se*. Pois um *join* consiste em um processo de união, em que estamos literalmente fundindo duas tabelas em uma só. Já a função `PROCV()`, é capaz de transportar apenas uma única coluna por tabela, logo, não é de sua filosofia, fundir as tabelas envolvidas. Por isso, se temos cinco colunas em uma tabela A, as quais desejamos levar até a tabela B, nós precisamos de cinco `PROCV()`'s diferentes no Excel, enquanto no R, precisamos de apenas um `inner_join()` para realizarmos tal ação.

Por último, vale destacar uma característica muito importante de um *join*, que é o seu processo de pareamento. Devido a essa característica, a ordem da linhas presentes em ambas as tabelas se torna irrelevante para o resultado. Por exemplo, veja na figura 6.2, um exemplo de *join*, onde a coluna ID representa a *key* entre as duas tabelas. Repare que as linhas na tabela à esquerda que se referem, por exemplo, aos indivíduos de ID 105, 107 e 108, se encontram em linhas diferentes na tabela à direita. Mesmo que esses indivíduos estejam em locais diferentes, a função responsável pelo *join*, vai realizar um pareamento entre as duas tabelas, antes de fundí-las. Dessa maneira, podemos nos certificar que as informações de cada indivíduo são corretamente posicionadas na tabela resultante.

6.4 Configurações sobre as colunas e keys utilizadas no join

Haverá momentos em que uma única coluna não será o bastante para identificarmos cada observação de nossa base. Por isso, teremos oportunidades em que devemos utilizar a combinação entre várias colunas, com o objetivo de formarmos uma *primary key* em nossa tabela.

Por exemplo, suponha que você trabalha diariamente com o registro de entradas no estoque de um supermercado. Imagine que você possua a tabela registro abaixo, que contém dados da seção de bebidas do estoque, e que apresentam o dia e mes em que uma determinada carga chegou ao estoque da empresa, além de uma descrição de seu conteúdo (*descricao*), seu valor de compra (*valor*) e as unidades inclusas (*unidades*).

Figura 6.2: Representação de um join entre duas tabelas

Fonte: Elaboração própria do autor.

```
registro <- tibble(
  dia = c(3, 18, 18, 25, 25),
  mes = c(2, 2, 2, 2, 3),
  ano = 2020,
  unidades = c(410, 325, 325, 400, 50),
  valor = c(450, 1400, 1150, 670, 2490),
  descricao = c("Fanta Laranja 350ml",
    "Coca Cola 2L", "Mate Couro 2L",
    "Kapo Uva 200ml", "Absolut Vodka 1L")
)

registro

## # A tibble: 5 x 6
##   dia  mes  ano unidades valor descricao
##   <dbl> <dbl> <dbl>   <dbl> <dbl>   <chr>
## 1     3     2 2020     410   450 Fanta Laranja 350ml
## 2    18     2 2020     325  1400 Coca Cola 2L
## 3    18     2 2020     325  1150 Mate Couro 2L
## 4    25     2 2020     400   670 Kapo Uva 200ml
## 5    25     3 2020      50  2490 Absolut Vodka 1L
```

Nessa tabela, as colunas dia, mes, ano, valor, unidades e descricao, sozinhas, são insuficientes para identificarmos cada carga registrada na tabela. Mesmo que, **atualmente**, cada valor presente na coluna descricao seja único, essa característica provavelmente não vai resistir por muito tempo. Pois o supermercado pode muito bem receber amanhã, por exemplo, uma outra carga de refrigeran-

tes de 2 litros da Mate Couro.

Por outro lado, a combinação dos valores presentes nas colunas dia, mes, ano, valor, unidades e descricao, pode ser o suficiente para criarmos um código de identificação único para cada carga. Por exemplo, ao voltarmos à tabela registro, podemos encontrar duas cargas que chegaram no mesmo dia 18, no mesmo mês 2, no mesmo ano de 2020, e trazendo as mesmas 325 unidades. Todavia, essas duas cargas, possuem descrições diferentes: uma delas incluía garrafas preenchidas com Coca Cola, enquanto a outra, continha Mate Couro. Concluindo, ao aliarmos as informações referentes a data de entrada (18/02/2020), as quantidades inclusas nas cargas (325 unidades), e as suas descrições (Coca Cola 2L e Mate Couro 2L), podemos enfim diferenciar essas duas cargas:

- 1) Uma carga que entrou no dia 18/02/2020, incluía 325 unidades de 2 litros de Coca Cola.
- 2) Uma carga que entrou no dia 18/02/2020, incluía 325 unidades de 2 litros de Mate Couro.

Como um outro exemplo, podemos utilizar as bases flights e weather, provenientes do pacote nycflights13. Perceba abaixo, que a base flights já possui um número grande colunas. Essa tabela apresenta dados diários, referentes a diversos voos que partiram da cidade de Nova York (EUA) durante o ano de 2013. Já a tabela weather, contém dados meteorológicos em uma dada hora, e em diversas datas do mesmo ano, e que foram especificamente coletados nos aeroportos da mesma cidade de Nova York.

```
library(nycflights13)
```

```
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <dbl>         <dbl>         <dbl>   <dbl>
## 1  2013     1     1     517             515             2     830
## 2  2013     1     1     533             529             4     850
## 3  2013     1     1     542             540             2     923
## 4  2013     1     1     544             545             -1    1004
## 5  2013     1     1     554             600             -6     812
## 6  2013     1     1     554             558             -4     740
## 7  2013     1     1     555             600             -5     913
## 8  2013     1     1     557             600             -3     709
## 9  2013     1     1     557             600             -3     838
## 10 2013     1     1     558             600             -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <dbl>,
## #   arr_delay <dbl>, carrier <chr>, flight <dbl>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

```
weather
```



```
## # A tibble: 26,115 x 15
##   origin year month   day hour temp dewp humid wind_dir wind_speed
##   <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>    <dbl>
## 1 EWR    2013     1     1     1  39.0  26.1  59.4     270     10.4
## 2 EWR    2013     1     1     2  39.0  27.0  61.6     250      8.06
## 3 EWR    2013     1     1     3  39.0  28.0  64.4     240     11.5
## 4 EWR    2013     1     1     4  39.9  28.0  62.2     250     12.7
## 5 EWR    2013     1     1     5  39.0  28.0  64.4     260     12.7
## 6 EWR    2013     1     1     6  37.9  28.0  67.2     240     11.5
## 7 EWR    2013     1     1     7  39.0  28.0  64.4     240     15.0
## 8 EWR    2013     1     1     8  39.9  28.0  62.2     250     10.4
## 9 EWR    2013     1     1     9  39.9  28.0  62.2     260     15.0
## 10 EWR   2013     1     1    10  41    28.0  59.6     260     13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>,
## #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dtm>
```

Ao aplicarmos um *join* entre essas tabelas, poderíamos analisar as características meterológicas que um determinado avião enfrentou ao levantar vô. Entretanto, necessitaríamos empregar ao menos cinco colunas diferentes para formarmos uma *key* adequada entre essas tabelas. Pois cada situação meterológica descrita na tabela *weather*, ocorre em um uma dada localidade, e em um horário específico de um determinado dia. Com isso, teríamos de utilizar as colunas: *year*, *month* e *day* para identificarmos a data correspondente a cada situação; mais a coluna *hour* para determinarmos o momento do dia em que essa situação ocorreu; além da coluna *origin*, que marca o aeroporto de onde cada vô partiu e, portanto, nos fornece uma localização no espaço geográfico para cada situação meteorológica.

Portanto, em todos os momentos em que você precisar utilizar um conjunto de colunas para formar uma *key*, como o caso das tabelas *weather* e *flights* acima, você deve fornecer um vetor contendo o nome dessas colunas para o argumento *by* da função de *join* que está utilizando, assim como no exemplo abaixo.

```
inner_join(
  flights,
  weather,
  by = c("year", "month", "day", "hour", "origin")
)
```

```
## # A tibble: 335,220 x 29
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>    <dbl>         <dbl>         <dbl>    <dbl>
## 1  2013     1     1      517             515           2       830
## 2  2013     1     1      533             529           4       850
## 3  2013     1     1      542             540           2       923
## 4  2013     1     1      544             545          -1      1004
## 5  2013     1     1      554             600          -6       812
```

```
## 6 2013 1 1 554 558 -4 740
## 7 2013 1 1 555 600 -5 913
## 8 2013 1 1 557 600 -3 709
## 9 2013 1 1 557 600 -3 838
## 10 2013 1 1 558 600 -2 753
## # ... with 335,210 more rows, and 22 more variables: sched_arr_time <dbl>,
## #   arr_delay <dbl>, carrier <chr>, flight <dbl>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour.x <dtm>, temp <dbl>, dewp <dbl>, humid <dbl>,
## #   wind_dir <dbl>, wind_speed <dbl>, wind_gust <dbl>, precip <dbl>,
## #   pressure <dbl>, visib <dbl>, time_hour.y <dtm>
```

Porém, a tabela `flights` já possui um número muito grande de colunas e, por essa razão, não conseguimos visualizar no resultado do `join`, as diversas colunas importadas da tabela `weather`. Sabemos que um `join` gera, por padrão, uma nova tabela contendo todas as colunas de ambas as tabelas utilizadas. Contudo, o exemplo acima demonstra que em certas ocasiões, o uso de muitas colunas pode sobrecarregar a sua visão e, com isso, dificultar o seu foco no que é de fato importante em sua análise.

Tendo isso em mente, haverá instantes em que você deseja trazer apenas algumas colunas de uma das tabelas envolvidas no `join`. Mas não há como alterarmos a natureza de um `join`, logo, todas as colunas de ambas as tabelas serão sempre incluídas em seu resultado. Por isso, o ideal é que você selecione as colunas desejadas de uma das tabelas antes de empregá-las em um `join`.

Ou seja, ao invés de fornecer a tabela completa à função, você pode utilizar ferramentas como `select()` ou `subsetting`, para extrair a parte desejada de uma das tabelas, e fornecer o resultado dessa seleção para a função `inner_join()`. Entretanto, **lembre-se sempre de incluir nessa seleção, as colunas que formam a *key* de seu `join`**. De outra forma, não se esqueça de incluir em sua seleção, as colunas que você proveu ao argumento `by`.

Por exemplo, supondo que você precisasse em seu resultado apenas das colunas `dep_time` e `dep_delay` da tabela `flights`, você poderia fornecer os comandos a seguir:

```
cols_para_key <- c(
  "year", # Coluna 1 para key
  "month", # Coluna 2 para key
  "day", # Coluna 3 para key
  "hour", # Coluna 4 para key
  "origin" # Coluna 5 para key
)

cols_desejadas <- c("dep_time", "dep_delay")

cols_c <- c(cols_para_key, cols_desejadas)

inner_join(
```

```

flights %>% select(all_of(cols_c)),
weather,
by = cols_para_key
)

## # A tibble: 335,220 x 17
##   year month   day hour origin dep_time dep_delay temp dewp humid
##   <int> <int> <int> <dbl> <chr>    <dbl>    <dbl> <dbl> <dbl> <dbl>
## 1  2013     1     1     5 EWR        517         2  39.0  28.0  64.4
## 2  2013     1     1     5 LGA        533         4  39.9  25.0  54.8
## 3  2013     1     1     5 JFK        542         2  39.0  27.0  61.6
## 4  2013     1     1     5 JFK        544        -1  39.0  27.0  61.6
## 5  2013     1     1     6 LGA        554        -6  39.9  25.0  54.8
## 6  2013     1     1     5 EWR        554        -4  39.0  28.0  64.4
## 7  2013     1     1     6 EWR        555        -5  37.9  28.0  67.2
## 8  2013     1     1     6 LGA        557        -3  39.9  25.0  54.8
## 9  2013     1     1     6 JFK        557        -3  37.9  27.0  64.3
## 10 2013     1     1     6 LGA        558        -2  39.9  25.0  54.8
## # ... with 335,210 more rows, and 7 more variables: wind_dir <dbl>,
## #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dtm>

## -----
## Ou por subsetting:
inner_join(
  flights[, cols_c],
  weather,
  by = cols_para_key
)

```

Antes de partirmos para a próxima seção, vale a pena comentar sobre um outro aspecto importante em um *join*. As colunas que formam a sua *key* **devem estar nomeadas da mesma maneira em ambas as tabelas**. Por exemplo, se nós voltarmos às tabelas `info` e `band_instruments`, e renomearmos a coluna `name` para `member` em uma das tabelas, um erro será levantado ao tentarmos aplicar novamente um *join* sobre as tabelas.

```

colnames(band_instruments)[1] <- "member"

inner_join(info, band_instruments, by = "name")

```

```

Erro: Join columns must be present in data.
x Problem with `name`.
Run `rlang::last_error()` to see where the error occurred.

```

Logo, ajustes são necessários sobre o argumento `by`, de forma a revelarmos para a função responsável pelo *join*, a existência dessa diferença existente entre os nomes dados às colunas que representam a

key entre as tabelas. Fazendo uso dos argumentos *x* e *y* como referências, para realizar esse ajuste, você deve igualar o nome dado à coluna da tabela *x* ao nome dado à coluna correspondente na tabela *y*, dentro de um vetor - `c()`, como está demonstrado abaixo.

```
inner_join(info, band_instruments, by = c("name" = "member"))
```

```
## # A tibble: 2 x 5
##   name band   born children plays
##   <chr> <chr>   <date>   <lgl>   <chr>
## 1 John Beatles 1940-09-10 TRUE    guitar
## 2 Paul Beatles 1942-06-18 TRUE    bass
```

6.5 Diferentes tipos de *join*

Portanto, um *join* busca construir uma união entre duas tabelas. Porém, podemos realizar essa união de diferentes formas, e até o momento, apresentei apenas uma de suas formas, o *inner join*, que é executado pela função `inner_join()`. Nesse método, o *join* mantém apenas as linhas que puderam ser encontradas em ambas as tabelas. Logo, se um indivíduo está presente na tabela A, mas não se encontra na tabela B, esse indivíduo será descartado em um *inner join* entre as tabelas A e B. Como foi destacado por Wickham e Grolemund (2017, p. 181), essa característica torna o *inner join* pouco apropriado para a maioria das análises, pois uma importante perda de observações pode ser facilmente gerada neste processo.

Com isso, nós podemos empregar tipos diferentes de *joins*, que são comumente chamados de *outer joins*, pois esses tipos buscam preservar as linhas de pelo menos uma das tabelas envolvidas no *join* em questão. Sendo eles:

1. `left_join()`: mantém todas as linhas da tabela definida no argumento *x*, ou a tabela à esquerda do *join*, mesmo que os indivíduos descritos nessa tabela não tenham sido encontrados em ambas as tabelas.
2. `right_join()`: mantém todas as linhas da tabela definida no argumento *y*, ou a tabela à direita do *join*, mesmo que os indivíduos descritos nessa tabela não tenham sido encontrados em ambas as tabelas.
3. `full_join()`: mantém todas as linhas de ambas as tabelas definidas nos argumentos *x* e *y*, mesmo que os indivíduos de uma dessas tabelas não tenham sido encontrados em ambas as tabelas.

Em todas as funções de *join* mostradas aqui, o primeiro argumento é chamado de *x*, e o segundo, de *y*, sendo esses os argumentos que definem as duas tabelas a serem utilizadas no *join*. Simplificadamente, a diferença entre `left_join()`, `right_join()` e `full_join()` reside apenas em quais linhas das tabelas utilizadas, são conservadas por essas funções no produto final do *join*. Como essas diferenças são simples, as descrições acima já lhe dão uma boa ideia de quais serão as linhas

conservadas em cada função. Todavia, darei a seguir, uma visão mais formal desses comportamentos, com o objetivo principal de fornecer uma segunda visão que pode, principalmente, facilitar a sua memorização do que cada função faz.

Para seguir esse caminho, é interessante que você tente interpretar um *join* a partir de uma perspectiva mais geral ou menos minuciosa do processo. Ao aplicarmos um *join* entre as tabelas A e B, estamos resumidamente, extraíndo as colunas da tabela B e as adicionando à tabela A (ou vice-versa). Com isso, temos nessa concepção, a **tabela fonte**, ou a tabela **de onde** as colunas são retiradas, e a **tabela destinatária**, ou a tabela **para onde** essas colunas são levadas. Portanto, segundo esse ponto de vista, o *join* possui sentido e direção, assim como um vetor em um espaço tridimensional. Pois o processo sempre parte da tabela fonte em direção a tabela destinatária. Dessa forma, em um *join*, ao construirmos uma nova tabela que representa a união entre duas tabelas, estamos basicamente extraíndo as colunas da tabela fonte e as incorporando à tabela destinatária.

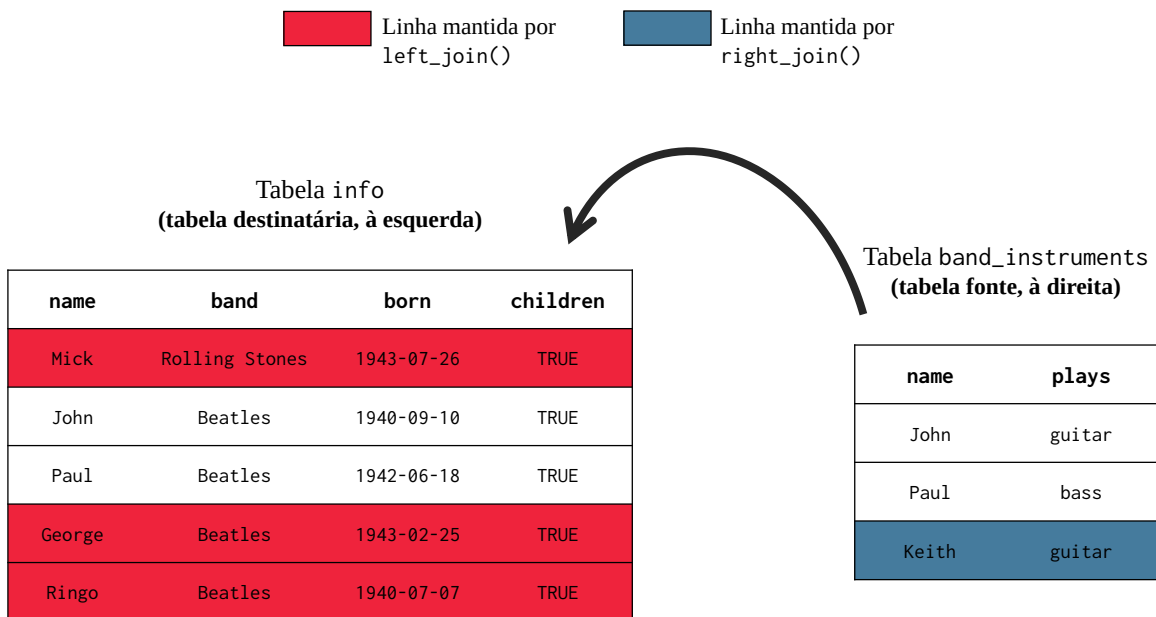
Com isso, eu quero criar a perspectiva, de que a tabela fonte e a tabela destinatária, ocupam lados do *join*, como na figura 6.3. Ou seja, por esse ângulo, estamos compreendendo o *join* como uma operação que ocorre sempre da direita para esquerda, ou um processo em que estamos sempre carregando um conjunto de colunas da tabela à direita em direção a tabela à esquerda. Se mesclarmos essa visão, com as primeiras descrições dos *outer joins* que fornecemos, temos que o argumento x corresponde a tabela destinatária, e o argumento y, a tabela fonte. Dessa maneira, a tabela destinatária (ou o argumento x) é sempre a tabela que ocupa o lado esquerdo do *join*, enquanto a tabela fonte (ou o argumento y) sempre se trata da tabela que ocupa o lado direito da operação.

Logo, a função `left_join()` busca manter as linhas da tabela destinatária (ou a tabela que você definiu no argumento x da função) intactas no resultado do *join*. Isso significa, que caso a função `left_join()` não encontre na tabela fonte, uma linha que corresponde a um certo indivíduo presente na tabela destinatária, essa linha será mantida no resultado final do *join*. Porém, como está demonstrado abaixo, em todas as situações em que a função não pôde encontrar esse indivíduo na tabela fonte, `left_join()` vai preencher as linhas correspondentes nas colunas que ele transferiu dessa tabela, com valores NA, indicando justamente que não há informações daquele respectivo indivíduo na tabela fonte.

```
left_join(info, band_instruments, by = "name")
```

```
## # A tibble: 5 x 5
##   name    band      born    children plays
##   <chr>  <chr>      <date>    <lgl>    <chr>
## 1 Mick   Rolling    1943-07-26 TRUE     <NA>
## 2 John   Beatles    1940-09-10 TRUE     guitar
## 3 Paul   Beatles    1942-06-18 TRUE     bass
## 4 George Beatles    1943-02-25 TRUE     <NA>
## 5 Ringo Beatles    1940-07-07 TRUE     <NA>
```

Em contrapartida, a função `right_join()` realiza justamente o processo contrário, ao manter as linhas da tabela fonte (ou a tabela que você forneceu ao argumento y). Por isso, para todas as linhas

Figura 6.3: As tabelas ocupam lados em um join

Fonte: Elaboração própria do autor.

da tabela fonte que se referem a um indivíduo não encontrado na tabela destinatária, `right_join()` acaba preenchendo os campos provenientes da tabela destinatária, com valores NA, indicando assim que a função não conseguiu encontrar mais dados sobre aquele indivíduo na tabela destinatária. Você pode perceber esse comportamento, pela linha referente ao músico Keith, que está disponível na tabela fonte, mas não na tabela destinatária.

```
right_join(info, band_instruments, by = "name")
```

```
## # A tibble: 3 x 5
##   name band   born children plays
##   <chr> <chr>   <date>   <lgl>   <chr>
## 1 John Beatles 1940-09-10 TRUE    guitar
## 2 Paul Beatles 1942-06-18 TRUE    bass
## 3 Keith <NA>    NA       NA      guitar
```

Por fim, a função `full_join()` executa o processo inverso da função `inner_join()`. Ou seja, se por um lado, a função `inner_join()` mantém as linhas de todos os indivíduos que puderam ser localizados em ambas as tabelas, por outro, a função `full_join()` não depende desses indivíduos aparecerem ou não em ambas as tabelas, ela sempre traz todos os indivíduos em seu resultado. Logo, `full_join()` mantém todas as linhas de ambas as tabelas. De certa forma, a função `full_join()` busca encontrar sempre o maior número possível de combinações entre as tabelas, e em todas as ocasiões que `full_join()` não encontra um determinado indivíduo, por exemplo, na tabela B, a função vai preencher os campos dessa tabela B com valores NA para as linhas desse indivíduo. Veja o exemplo abaixo.

```
full_join(info, band_instruments, by = "name")
```

```
## # A tibble: 6 x 5
##   name band   born children plays
##   <chr> <chr>   <date>   <lgl>   <chr>
## 1 Mick Rolling Stones 1943-07-26 TRUE    <NA>
## 2 John Beatles      1940-09-10 TRUE    guitar
## 3 Paul Beatles      1942-06-18 TRUE    bass
## 4 George Beatles     1943-02-25 TRUE    <NA>
## 5 Ringo Beatles      1940-07-07 TRUE    <NA>
## 6 Keith <NA>          NA       NA      guitar
```

Como o primeiro `data.frame` fornecido à função `*_join()`, será na maioria das situações, a sua principal tabela de trabalho, o ideal é que você adote o `left_join()` como o seu padrão de *join* (WICKHAM; GROLEMUND, 2017). Pois dessa maneira, você evita uma possível perda de observações em sua tabela mais importante.

6.6 Relações entre *keys*: *primary keys* são menos comuns do que você pensa

Na seção **Dados relacionais e o conceito de *key***, nós estabelecemos que variáveis com a capacidade de identificar unicamente cada observação de sua base, podem ser caracterizadas como *primary keys*. Mas para que essa característica seja verdadeira para uma dada variável, os seus valores não podem se repetir ao longo da base, e isso não acontece com tanta frequência na realidade.

Como exemplo, podemos voltar ao *join* entre as tabelas *flights* e *weather* que mostramos na seção **Configurações sobre as colunas e *keys* utilizadas no *join***. Para realizarmos o *join* entre essas tabelas, nós utilizamos as colunas *year*, *month*, *day*, *hour* e *origin* como *key*. Porém, a forma como descrevemos essas colunas na seção passada, ficou subentendido que a combinação entre elas foi capaz de formar uma *primary key*. Bem, porque não conferimos se essas colunas assumem de fato esse atributo:

```
flights %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
```

```
## # A tibble: 18,906 x 6
##   year month   day hour origin     n
##   <int> <int> <int> <dbl> <chr>  <int>
## 1  2013     1     1     5 EWR      2
## 2  2013     1     1     5 JFK      3
## 3  2013     1     1     6 EWR     18
## 4  2013     1     1     6 JFK     17
## 5  2013     1     1     6 LGA     17
## 6  2013     1     1     7 EWR     12
## 7  2013     1     1     7 JFK     16
## 8  2013     1     1     7 LGA     21
## 9  2013     1     1     8 EWR     20
## 10 2013     1     1     8 JFK     23
## # ... with 18,896 more rows
```

Como podemos ver acima, há diversas combinações entre as cinco colunas que se repetem ao longo da base. Com isso, podemos afirmar que a combinação entre as colunas *year*, *month*, *day*, *hour* e *origin* não formam uma *primary key*. Perceba abaixo, que o mesmo vale para a tabela *weather*:

```
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
```

```
## # A tibble: 3 x 6
##   year month   day hour origin     n
```



```
##    <int> <int> <int> <int> <chr>  <int>
## 1  2013    11     3     1 EWR      2
## 2  2013    11     3     1 JFK      2
## 3  2013    11     3     1 LGA      2
```

Portanto, circunstâncias em que não há uma *primary key* definida entre duas tabelas, são comuns, inclusive em momentos que você utiliza a combinação de todas as colunas disponíveis em uma das tabelas para formar uma *key*. Com isso, eu quero destacar principalmente, que não há problema algum em utilizarmos *foreign keys* em *joins*.

Logo, você deve definir a *key* mais apropriada para o seu *join*, baseado no seu conhecimento sobre esses dados, e não de forma a procurar por colunas de mesmo nome em ambas as colunas (WICKHAM; GROLEMUND, 2017). Durante esse processo, nós não estamos perseguindo *primary keys* de maneira obsessiva, mas sim, pesquisando por relações verdadeiras e lógicas entre as tabelas.

Por exemplo, no caso das tabelas *flights* e *weather*, utilizamos as colunas *year*, *month*, *day*, *hour* e *origin* como *key*, pelo fato de que eventos climáticos ocorrem um dado momento (*hour*) de um dia específico (*year*, *month* e *day*), além de geralmente se restringir a uma dada região geográfica (*origin*). Curiosamente, essas colunas não foram suficientes para produzirmos uma *primary key*, mas foi o suficiente para representarmos uma conexão lógica entre as tabelas *flights* e *weather*.

Assim sendo, qualquer que seja o tipo de *key* empregado, o processo de *join* irá ocorrer exatamente da mesma forma. Porém, o tipo que a *key* assume em cada tabela pode alterar as combinações geradas no resultado do *join*. Como temos duas tabelas em cada *join*, temos três possibilidades de relação entre as *keys* de cada tabela: 1) *primary key* → *primary key*; 2) *primary key* → *foreign key*; 3) *foreign key* → *foreign key*. Ou seja, em cada uma das tabelas envolvidas em um *join*, as colunas a serem utilizadas como *key* podem se caracterizar como uma *primary key* ou como uma *foreign key*.

Como exemplo, o *join* formado pelas tabelas *info* e *band_instruments*, possui uma relação de *primary key* → *primary key*. Pois a coluna *name* é uma *primary key* em ambas as tabelas. Por outro lado, o *join* formado pelas tabelas *flights* e *weather*, possui uma relação de *foreign key* → *foreign key*, visto que as cinco colunas utilizadas como *key* não são capazes de identificar unicamente cada observação nas duas tabelas, como comprovamos acima.

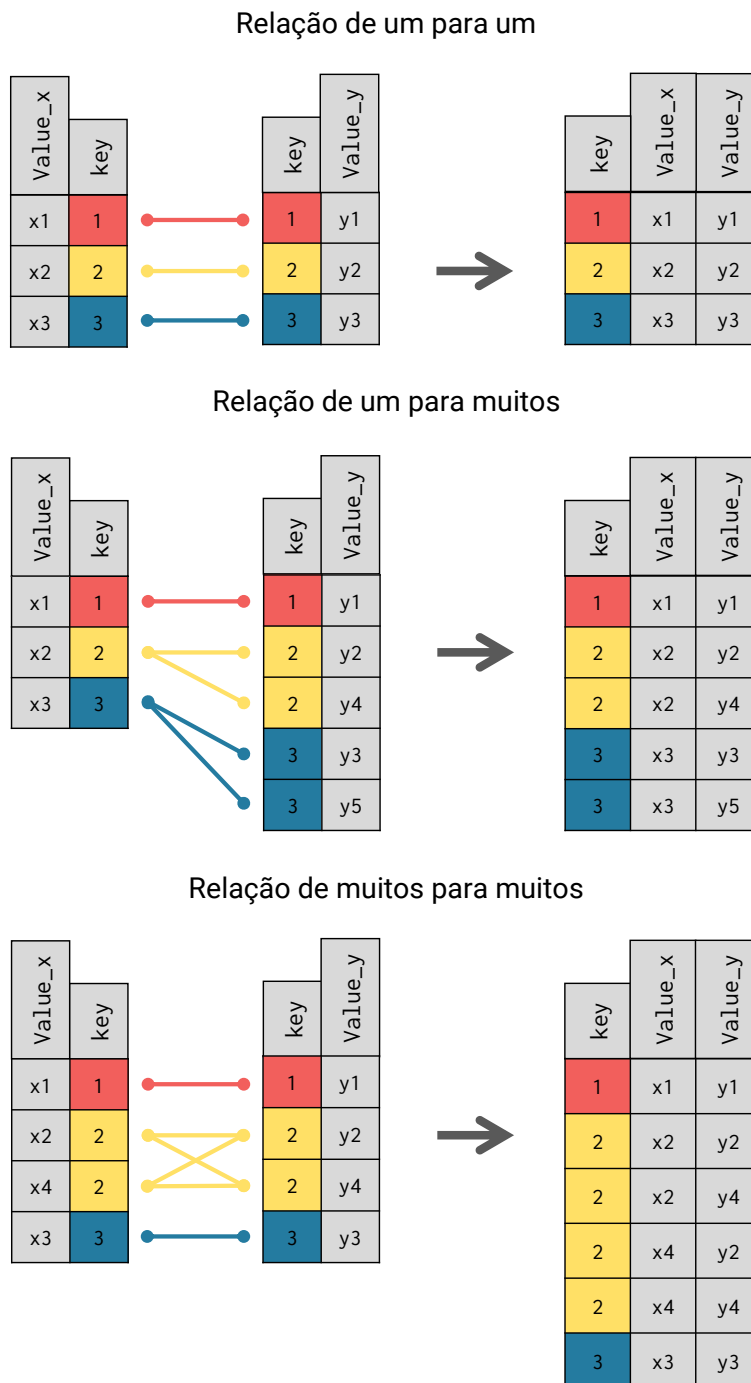
Com isso, temos a opção de compreendermos a relação entre as *keys*, como uma relação de quantidade de cópias, fazendo referência direta ao fato de que uma *primary key* não possui valores repetidos ao longo da base, enquanto o mesmo não pode ser dito de uma *foreign key*. Logo, uma relação *primary key* → *primary key* pode ser identificada como uma relação de **um para um**, pois sempre vamos contar com uma única chave para cada observação em ambas as tabelas. Para mais, podemos interpretar uma relação *primary key* → *foreign key*, como uma relação de **um para muitos**, pois para cada chave única presente em uma das tabelas, podemos dispor de múltiplas irmãs gêmeas presentes na outra tabela.

Deste modo, se tivermos uma relação *foreign key* → *foreign key*, ou uma relação de **muitos para**

muitos, para cada conjunto de *keys* repetidas em ambas as tabelas, todas as possibilidades de combinação são geradas. Em outras palavras, nesse tipo de relação, o resultado do *join* será uma produto cartesiano como demonstrado pela figura 6.4.

Relações de um para um são raras e, por essa razão, vamos mais comumente possuir uma relação de um para muitos em nossas tabelas, onde nesse caso, as *primary keys* são replicadas no resultado do *join*, para cada repetição de sua *key* correspondente na outra tabela, como pode ser visto na figura 6.4.

Figura 6.4: Resumo das relações possíveis entre keys, inspirado em Wickham e Grolemund (2017)



Fonte: Elaboração própria do autor. Inspirado em [WICKHAM; GROLEMUND, 2017](#), p. 182.

Exercícios

O primeiro exercício desse capítulo, envolve duas tabelas publicadas na semana 11 do projeto Tidy Tuesday em 2020. Mais especificamente, as tabelas `tuition_cost` e `salary_potential`. A tabela `tuition_cost` descreve os custos de um curso de graduação em diferentes universidades dos EUA. Em contrapartida, a tabela `salary_potential` fornece uma estimativa do salário pontencial que um diploma de graduação de diversas universidades dos EUA podem fornecer a um profissional.

No Brasil, as faculdades privadas geralmente cobram por uma mensalidade fixa que abrange todos os custos mínimos. Já algumas universidades privadas, tendem a usar um sistema mais complexo, onde uma mensalidade base é cobrada, além de taxas por aulas práticas (para cobrir gastos com o uso de equipamentos) e taxas por matéria matriculada. Em outras palavras, um aluno de uma universidade privada brasileira que se matricula, por exemplo, em 4 matérias num dado semestre, geralmente paga um valor mensal que segue a estrutura: mensalidade base + taxa por aula prática (se houver alguma aula prática) + $(4 \times \text{taxa por matrícula})$.

Por outro lado, as universidades americanas possuem um sistema mais complexo de cobrança. Primeiro, a maior parte dos estudantes americanos optam por morar e se alimentar nos alojamentos da universidade, ao invés de se manterem na casa dos pais. A universidade cobra uma taxa específica para esses estudantes, que busca pagar justamente os custos deste alojamento e de sua alimentação. Tal custo é geralmente denominado de *room and board fees*. Segundo, universidades americanas cobram principalmente pelo seu ensino'' (e alguns outros serviços) e, por isso, a maior parte de seus preços envolvem o que chamamos de *\emph{tuition fees}*'' (ou “taxa de ensino”). Terceiro, os valores divulgados pelas universidades são geralmente anuais, logo, se o *tuition fees* (ou *room and board fees*) de uma universidade qualquer é de \$25 mil, isso significa que um curso de 4 anos nessa universidade custaria em torno de \$100 mil.

Portanto, as universidades americanas cobram, em geral, dois tipos de custos diferentes (*room and board fees* e *tuition fees*) e, esses custos são em sua maioria, anuais. Grande parte dos alunos acabam pagando ambos desses custos, logo, esses custos somados representam, para grande parte da população, o custo total por ano de uma universidade nos EUA.

Para mais, as universidades americanas também cobram taxas de ensino (*tuition fees*) diferentes de acordo com o estado em que o aluno reside. Ou seja, uma universidade que está sediada no estado do Texas vai cobrar uma taxa mais barata para os alunos que moram no estado do Texas. Porém, os alunos que são originalmente de outros estados, e estão vindo para essa universidade vão pagar taxas maiores.

Exercício 1

Suponha que você esteja interessado em realizar um curso de graduação em alguma das universidades descritas na tabela `tuition_cost`. Como você provavelmente não mora nos Estados Unidos, considere os custos referentes a alunos *out of state* em seus cálculos. Vale também ressaltar que os salários estimados na tabela `salary_potential`, assim como os custos na tabela `tuition_cost`, são anuais. Com base nas estimativas de salário presentes na tabela `salary_potential` e, com base

nos custos descritos na tabela `tuition_cost`, tente calcular (para cada universidade) o tempo de trabalho necessário (após a graduação) para pagar pelo investimento que você aplicou no curso de graduação.

```
library(tidyverse)

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

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

Exercício 2

Todos os itens abaixo envolvem as tabelas `consumidores` e `vendedores`, alguns itens serão teóricos, outros, vão lhe requisitar o cálculo de alguma informação. Como esses cálculos envolvem as informações de ambas as tabelas, você será obrigado a aplicar um *join* entre elas para realizá-lo:

```
library(tidyverse)

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

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

- A) Quais colunas representam as *keys* em ambas as tabelas?
- B) Na tabela `consumidores`, quais colunas representam uma *primary key*, e quais representam uma *foreign key*?
- C) Descubra o número de cidades nas quais cada vendedor atendeu os seus clientes.

Exercício 3

Dado que você tenha importado as tabelas `filmes` e `filmes_receita` abaixo para o seu R, e, tendo em mente o que vimos nesse capítulo, explique porque o comando de *join* abaixo não funciona sobre essas tabelas. Dado que você encontre e explique o que está errado, como você ajustaria esse comando para que ele funcione normalmente?

```
library(tidyverse)

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

```
arquivo1 <- "filmes_dados.csv"
arquivo2<- "filmes_receita.csv"

filmes <- read_csv2(paste0(github, pasta, arquivo1))
filmes_receita <- read_csv2(paste0(github, pasta, arquivo2))

### Porque esse comando de join
### abaixo não funciona?
filmes %>%
  left_join(
    filmes_receita
  )
```

Capítulo 7

Tidy Data: Uma abordagem para organizar os seus dados

7.1 Introdução e pré-requisitos

Em qualquer análise, o formato no qual os seus dados se encontram, é muito importante. O que vamos discutir neste capítulo, será como reformatar as suas tabelas, corrigir valores não disponíveis, ou “vazios” que se encontram no formato incorreto, ou então, como preencher as suas colunas que estão incompletas de acordo com um certo padrão.

Você rapidamente descobre a importância que o formato de sua tabela carrega para o seu trabalho, na medida em que você possui pensamentos como: “Uhhh...se essa coluna estivesse na forma x, eu poderia simplesmente aplicar a função `y()` e todos os meus problemas estariam resolvidos”; ou então: “Se o Arnaldo não tivesse colocado os totais junto dos dados desagregados, eu não teria todo esse trabalho!”; ou talvez: “Qual é o sentido de colocar o nome dos países nas colunas? Assim fica muito mais difícil de acompanhar os meus dados!”.

Para corrigir o formato das nossas tabelas, vamos utilizar neste capítulo as funções do pacote `tidyr` que está incluso no `tidyverse`. Pelo próprio nome do pacote (*tidy*, que significa “arrumar”), já sabemos que ele inclui diversas funções que tem como propósito, organizar os seus dados. Portanto, lembre-se de chamar pelo pacote (seja pelo *tidyr* diretamente, ou pelo *tidyverse*) antes de prosseguir:

```
library(tidyverse)
## Ou
library(tidyr)
```

7.2 O que é *tidy data*?

Em geral, nós passamos grande parte do tempo, reorganizando os nossos dados, para que eles fiquem em um formato adequado para a nossa análise. Logo, aprender técnicas que facilitem o seu trabalho

nesta atividade, pode economizar uma grande parte de seu tempo.

Isso é muito importante, pois uma base de dados que está bagunçada, é em geral bagunçada em sua própria maneira. Como resultado, cada base irá exigir um conjunto de operações e técnicas diferentes das outras bases, para que ela seja arrumada. Algumas delas, vão enfrentar problemas simples de serem resolvidos, já outras, podem estar desarrumadas em um padrão não muito bem definido, e por isso, vão dar mais trabalho para você. Por essas razões, aprender técnicas voltadas para esses problemas, se torna uma atividade necessária.

“Tidy datasets are all alike, but every messy dataset is messy in its own way”.
(WICKHAM, 2014, p. 2)

Toda essa problemática, ocorre não apenas pelo erro humano, mas também porque podemos representar os nossos dados de diversas maneiras em uma tabela. Sendo que essas maneiras, podem tanto facilitar muito o seu trabalho, quanto tornar o trabalho de outros, num inferno. Veja por exemplo, as tabelas abaixo. Ambas, apresentam os mesmos dados, mas em estruturas diferentes.

table2

```
## # A tibble: 12 x 4
##   country    year type      count
##   <chr>      <int> <chr>    <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases      2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases      37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases      80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases      212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases      213766
## 12 China      2000 population 1280428583
```

table3

```
## # A tibble: 6 x 3
##   country    year rate
## * <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China      2000 213766/1280428583
```


Antes de partirmos para a prática, vou lhe fornecer uma base teórica que irá sustentar as suas decisões sobre como padronizar e estruturar os seus dados. Eu expliquei anteriormente, que o tidyverse é um conjunto de pacotes que dividem uma mesma filosofia. Isso significa, que esses pacotes possuem uma conexão forte entre si. Por exemplo, as funções desses pacotes, retornam os seus resultados em tibble's, e todas as suas funções foram construídas de forma a trabalharem bem com o operador *pipe* (%>%). Todas essas funções também foram projetadas seguindo as melhores práticas e técnicas em análise de dados. Sendo uma dessas práticas, o que é comumente chamado na comunidade de *tidy data*.

O conceito de *tidy data* foi definido por Wickham (2014), e remete a forma como você está guardando os dados em sua tabela. Eu não estou dizendo aqui que todas as funções do tidyverse que apresentei até aqui, trabalham apenas com *tidy data*, mas sim, que essas funções são mais eficientes com essa estrutura *tidy*. Uma base de dados que está no formato *tidy*, compartilha das três seguintes características:

- 1) Cada variável de sua tabela, deve possuir a sua própria coluna.
- 2) Cada observação de sua tabela, deve possuir a sua própria linha.
- 3) Cada valor de sua tabela, deve possuir a sua própria célula.

Eu posso pressupor que essas definições acima, já são claras o suficiente para que você entenda o que são dados *tidy*. Porém, deixar as coisas no ar, é com certeza uma prática tão ruim quanto incluir totais junto de seus dados desagregados. Por isso, vou passar os próximos parágrafos definindo com maior precisão cada parte que compõe essas características.

Primeiro, vou definir o que quero dizer exatamente com linhas, colunas e células de sua tabela. Abaixo temos uma representação de uma base qualquer. O interesse nessa representação, não se trata dos valores e nomes inclusos nessa tabela, mas sim as áreas sombreadas dessa tabela, que estão lhe apresentando cada um dos componentes supracitados.

Agora, vamos definir o que são variáveis, observações e valores. Você já deve ter percebido, que toda base de dados, possui uma unidade básica que está sendo descrita ao longo dela. Ou seja, toda base lhe apresenta dados sobre um grupo específico (ou uma amostra) de algo. Esse algo pode ser um conjunto de municípios, empresas, sequências genéticas, animais, clientes, realizações de um evento estocástico, dentre outros.

Logo, se a minha base contém dados sobre os municípios do estado de Minas Gerais (MG), cada um desses municípios são uma observação de minha base. Ao dizer que cada observação deve possuir a sua própria linha, eu estou dizendo que todas as informações referentes a um município específico, devem estar em uma única linha. Em outras palavras, cada uma das 853 (total de municípios em MG) linhas da minha base, contém os dados de um município diferente do estado.

Entretanto, se a minha base descreve a evolução do PIB desses mesmos municípios nos anos de 2010 a 2020, eu não possuo mais um valor para cada município, ao longo da base. Neste momento, eu possuo 10 valores diferentes, para cada município, e mesmo que eu ainda esteja falando dos mesmos municípios, a unidade básica da minha base, se alterou. Cada um desses 10 valores, representa uma

Figura 7.1: Definindo colunas, linhas e células de uma tabela

Coluna			Linha			Célula		
ID	Idade	Peso	ID	Idade	Peso	ID	Idade	Peso
1	20	80	1	20	80	1	20	80
2	18	78	2	18	78	2	18	78
3	10	50	3	10	50	3	10	50
4	12	65	4	12	65	4	12	65
5	34	89	5	34	89	5	34	89
6	22	63	6	22	63	6	22	63
7	27	69	7	27	69	7	27	69

Fonte: Elaboração própria do autor.

observação do PIB deste município em um ano distinto. Logo, cada um desses 10 valores para cada município, deve possuir a sua própria linha. Se o estado de Minas Gerais possui 853 municípios diferentes, isso significa que nossa base deveria ter $10 \times 853 = 8.530$ linhas. Por isso, é importante que você preste atenção em seus dados, e identifique qual é a unidade básica que está sendo tratada.

Agora, quando eu me referir as variáveis de sua base, eu geralmente estou me referindo as colunas de sua base, porque ambos os termos são sinônimos em análises de dados. Porém, alguns cuidados são necessários, pois as variáveis de sua base podem não se encontrar nas colunas de sua tabela. Como eu disse anteriormente, há diversas formas de representar os seus dados, e por isso, há diversas formas de alocar os componentes de seus dados ao longo de sua tabela.

Uma variável de sua base de dados, não é apenas um elemento que (como o próprio nome dá a entender) varia ao longo de sua base, mas é um elemento que lhe apresenta uma característica das suas observação. Cada variável me descreve uma característica (cor de pele, população, receita, ...) de cada observação (pessoa, município, empresa, ...) da minha base. O que é ou não, uma característica de sua unidade básica, irá depender de qual é essa unidade básica que está sendo descrita na base.

A população total, é uma característica geralmente associada a regiões geográficas (municípios, países, etc.), já a cor de pele pode ser uma característica de uma amostra de pessoas entrevistadas em uma pesquisa de campo (como a PNAD contínua), enquanto o número total de empresas é uma característica associada a setores da atividade econômica (CNAE - Classificação Nacional de Atividades Econômicas).

Por último, os valores de sua base, correspondem aos registros das características de cada observa-

ção de sua base. Como esse talvez seja o ponto mais claro e óbvio de todos, não vou me prolongar mais sobre ele. Pois as três características de *tidy data* que citamos anteriormente são interrelacionadas, de forma que você não pode satisfazer apenas duas delas. Logo, se você está satisfazendo as duas primeiras, você não precisa se preocupar com a característica que diz respeito aos valores.

Figura 7.2: Três propriedades que caracterizam o formato *tidy data*



Fonte: Elaboração própria do autor. Inspirado em [WICKHAM; GROLEMUND, 2017](#), p. 149.

Portanto, sempre inicie o seu trabalho, identificando a unidade básica de sua base. Em seguida, tente encontrar quais são as suas variáveis, ou as características dessa unidade básica que estão sendo descritas na base. Após isso, basta alocar cada variável em uma coluna, e reservar uma linha para cada observação diferente de sua base, que você automaticamente estará deixando uma célula para cada valor da base.

7.2.1 Será que você entendeu o que é tidy data?

Nessa seção vamos fazer um teste rápido, para saber se você entendeu o que é uma tabela no formato *tidy*. Olhe por algum tempo para os exemplos abaixo, e reflita sobre qual dessas tabelas está no formato *tidy*. Tente também descobrir quais são os problemas que as tabelas “não *tidy*” apresentam, ou em outras palavras, qual das três definições que apresentamos anteriormente, que essas tabelas “não *tidy*” acabam rompendo.

table1

```
## # A tibble: 6 x 4
##   country    year cases population
##   <chr>      <int> <int>      <int>
```

```
## 1 Afghanistan 1999 745 19987071
## 2 Afghanistan 2000 2666 20595360
## 3 Brazil 1999 37737 172006362
## 4 Brazil 2000 80488 174504898
## 5 China 1999 212258 1272915272
## 6 China 2000 213766 1280428583
```

table2

```
## # A tibble: 12 x 4
##   country      year type      count
##   <chr>      <int> <chr>    <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases      2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil 1999 cases      37737
## 6 Brazil 1999 population 172006362
## 7 Brazil 2000 cases      80488
## 8 Brazil 2000 population 174504898
## 9 China 1999 cases      212258
## 10 China 1999 population 1272915272
## 11 China 2000 cases      213766
## 12 China 2000 population 1280428583
```

table3

```
## # A tibble: 6 x 3
##   country      year rate
##   * <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil 1999 37737/172006362
## 4 Brazil 2000 80488/174504898
## 5 China 1999 212258/1272915272
## 6 China 2000 213766/1280428583
```

Como eu disse anteriormente, a primeira coisa que você deve fazer, é identificar a unidade básica que está sendo tratada na tabela. Nos exemplos acima, essas tabelas dizem respeito à dados de três países (Brasil, China e Afeganistão) em dois anos diferentes (1999 e 2000). Logo, a nossa tabela possui $3 \times 2 = 6$ observações diferentes. Se uma das regras, impõe que todas as linhas devem possuir informações de uma única observação, a nossa tabela deveria possuir 6 linhas. Com isso, nós já sabemos que algo está errado com a tabela 2, pois ela possui o dobro de linhas.

Na verdade, o problema na tabela 2 é que ela está quebrando a regra de que cada variável na tabela deve possuir a sua própria coluna. Por causa dessa regra, a tabela 2 acaba extrapolando o número de

linhas necessárias. Olhe para as colunas `type` e `count`. A coluna `count` lhe apresenta os principais valores que estamos interessados nessa tabela. Porém, a coluna `type`, está lhe apresentando duas variáveis diferentes.

Lembre-se de que variáveis, representam características da unidade básica de sua tabela. No nosso caso, essa unidade básica são dados anuais de países, logo, `cases` e `population`, são variáveis ou características diferentes desses países. Uma dessas variáveis está lhe apresentando um dado demográfico (população total), já a outra, está lhe trazendo um indicador epidemiológico (número de casos de alguma doença). Por isso, ambas variáveis deveriam possuir a sua própria coluna.

Ok, mas e as tabelas 1 e 3? Qual delas é a *tidy*? Talvez, para responder essa pergunta, você deveria primeiro procurar pela tabela “não *tidy*”. Veja a tabela 3, e se pergunte: “onde se encontram os valores de população e de casos de cada país nessa tabela?”. Ao se fazer essa pergunta, você provavelmente já irá descobrir qual é o problema nessa tabela.

A tabela 3, também rompe com a regra de que cada variável deve possuir a sua própria coluna. Pois o número de casos e a população total, estão guardados em uma mesma coluna! Ao separar os valores de população e de número de casos na tabela 3, em duas colunas diferentes, você chega na tabela 1, que é um exemplo de tabela *tidy*, pois agora todas as três definições estão sendo respeitadas.

7.2.2 Uma breve definição de formas

Apenas para que os exemplos das próximas seções, fiquem mais claros e fáceis de se visualizar mentalmente, vou definir dois formatos gerais que a sua tabela pode assumir, que são: *long* (longa) e *wide* (larga)¹. Ou seja, qualquer que seja a sua tabela, ela vai em geral, estar em algum desses dois formatos, de uma forma ou de outra.

Esses termos (*long* e *wide*) são bem descritivos por si só. A ideia é que se uma tabela qualquer, está no formato *long*, ela adquire um aspecto visual de longa, ou em outras palavras, visualmente ela aparenta ter muitas linhas, e poucas colunas. Já uma tabela que está no formato *wide*, adquire um aspecto visual de larga, como se essa tabela possuísse mais colunas do que o necessário, e poucas linhas. Perceba pelos exemplos apresentados na figura 7.3, que estamos apresentando exatamente os mesmos dados, eles apenas estão organizados de formas diferentes ao longo das duas tabelas.

7.3 Operações de pivô

As operações de pivô são as principais operações que você irá utilizar para reformatar a sua tabela. O que essas operações fazem, é basicamente alterar as dimensões de sua tabela, ou dito de outra maneira, essas operações buscam transformar colunas em linhas, ou vice-versa. Para exemplificar essas operações, vamos utilizar as tabelas que vem do próprio pacote `tidyr`. Logo, se você chamou pelo `tidyverse` através de `library()`, você tem acesso a tabela abaixo. Basta chamar no console pelo objeto `relig_income`.

¹Esses são termos comuns na comunidade de R, mas estes formatos também são conhecidos, ou chamados por *indexed data* (*long*) e por *cartesian data* (*wide*).

Figura 7.3: Formas gerais que a sua tabela pode adquirir*Long (Longa)*

Nome	Variável	Valor
Ana	Peso	61
Ana	Idade	20
Ana	Altura	1,71
Eduardo	Peso	90
Eduardo	Idade	18
Eduardo	Altura	1,82
Isabela	Peso	68
Isabela	Idade	19
Isabela	Altura	1,64
Leticia	Peso	82
Leticia	Idade	23
Leticia	Altura	1,88
Paulo	Peso	78
Paulo	Idade	27
Paulo	Altura	1,75

Wide (Larga)

Nome	Idade	Peso	Altura
Ana	20	61	1,71
Eduardo	18	90	1,82
Isabela	19	68	1,64
Leticia	23	82	1,88
Paulo	27	78	1,75

Fonte: Elaboração própria do autor.

```
relig_income
```

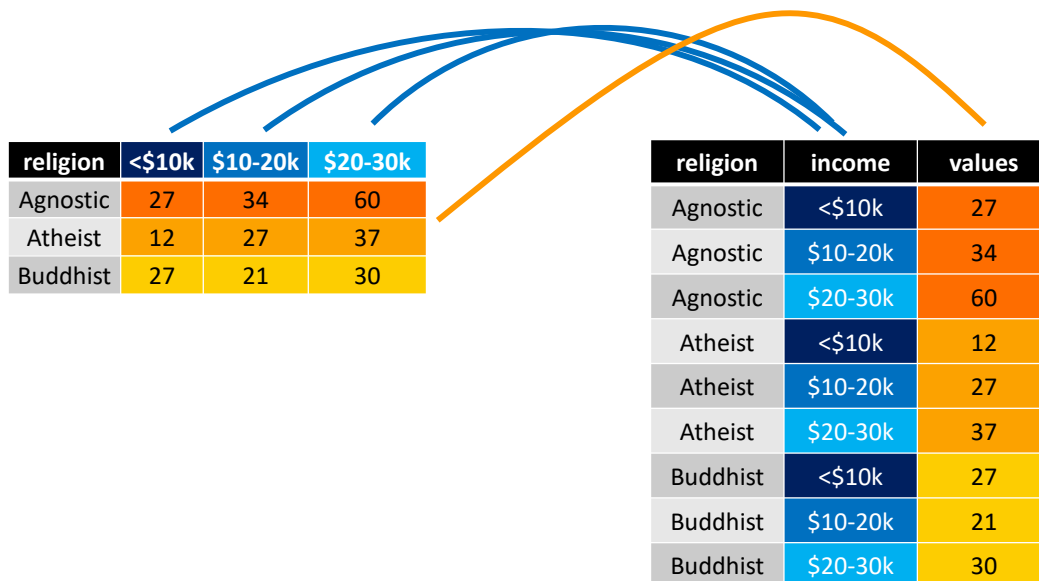
```
## # A tibble: 18 x 11
##   religion      `<$10k` ` $10-20k` ` $20-30k` ` $30-40k` ` $40-50k` ` $50-75k`
##   <chr>         <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Agnostic      27      34      60      81      76     137
## 2 Atheist       12      27      37      52      35      70
## 3 Buddhist      27      21      30      34      33      58
## 4 Catholic     418     617     732     670     638    1116
## 5 Don't know/re~ 15      14      15      11      10      35
## 6 Evangelical P~ 575     869    1064     982     881    1486
## 7 Hindu          1       9       7       9      11      34
## 8 Historically ~ 228     244     236     238     197     223
## 9 Jehovah's Wit~ 20      27      24      24      21      30
## 10 Jewish        19      19      25      25      30      95
## 11 Mainline Prot  289     495     619     655     651    1107
## 12 Mormon        29      40      48      51      56     112
## 13 Muslim         6       7       9      10       9      23
## 14 Orthodox      13      17      23      32      32      47
## 15 Other Christi~  9       7      11      13      13      14
## 16 Other Faiths   20      33      40      46      49      63
## 17 Other World R~  5       2       3       4       2       7
## 18 Unaffiliated  217     299     374     365     341     528
## # ... with 4 more variables: $75-100k <dbl>, $100-150k <dbl>, >150k <dbl>,
## #   Don't know/refused <dbl>
```

Essa tabela está nos apresentando o salário médio de pessoas pertencentes a diferentes religiões. Veja que em cada coluna dessa tabela, você possui os dados de um nível (ou faixa) salarial específico. Essa é uma estrutura que pode ser fácil e intuitiva em alguns momentos, mas certamente irá trazer limites importantes para você dentro do R. Devido a especialidade que o R possui sobre operações vetorizadas, o ideal seria transformarmos essa tabela para o formato *tidy*.

A unidade básica dessa tabela, são os grupos religiosos, e a faixa salarial representa uma característica desses grupos. Há diferentes níveis salariais na tabela, que estão sendo distribuídos ao longo de diferentes colunas. Tendo em vista isso, uma das regras não está sendo respeitada, pois todos esses diferentes níveis salariais, representam uma única característica, ou em outras palavras, eles transmitem o mesmo tipo de informação, que é um nível salarial daquele grupo religioso. Por isso, todas essas características da tabela, deve estar em uma única coluna. Em uma representação visual resumida, é isso o que precisamos fazer:

Por isso, quando você estiver em um momento como este, em que você deseja reformatar a sua tabela, ou em outras palavras, transformar as suas linhas em colunas, ou vice-versa, você está na verdade, procurando realizar uma operação de pivô.

Nestas situações, você deve primeiro pensar como a sua tabela ficará, após a operação de pivô que você deseja aplicar. Ou seja, após essa operação, a sua tabela ficará com mais linhas/colunas? Ou

Figura 7.4: Representação de uma operação de pivô

Fonte: Elaboração própria do autor.

menos linhas/colunas? Em outras palavras, você precisa identificar se você deseja tornar a sua tabela mais longa (aumentar o número de linhas, e reduzir o número de colunas), ou então, se você deseja torná-la mais larga (reduzir o número de linhas, e aumentar o número de colunas).

7.3.1 Adicionando linhas à sua tabela com `pivot_longer()`

Atualmente, a tabela `relig_income` possui poucas linhas e muitas colunas, e por isso, ela adquire um aspecto visual de “larga”. Como eu disse, seria muito interessante para você, que transformasse essa tabela, de modo a agrupar as diferentes faixas de níveis salariais em menos colunas. Logo, se estamos falando em reduzir o número de colunas, estamos querendo alongar a base, ou dito de outra forma, aumentar o número de linhas da base. Para fazermos isso, devemos utilizar a função `pivot_longer()`.

Essa função possui três argumentos principais: 1) `cols`, os nomes das colunas que você deseja transformar em linhas; 2) `names_to`, o nome da nova coluna onde serão alocados os nomes, ou os rótulos das colunas que você definiu em `cols`; 3) `values_to`, o nome da nova coluna onde serão alocados os valores da sua tabela, que se encontram nas colunas que você definiu em `cols`. Como nós queremos transformar todas as colunas da tabela `relig_income`, que contém faixas salariais, eu posso simplesmente colocar no argumento `cols`, um símbolo de menos antes do nome da coluna `religion`, que é a única coluna da tabela, que não possui esse tipo de informação. Ou seja, dessa forma, eu estou dizendo à `pivot_longer()`, para transformar todas as colunas (exceto a coluna `religion`).

```
relig_income %>%
  pivot_longer(
    cols = -religion,
    names_to = "income",
    values_to = "values"
  )
```

```
## # A tibble: 180 x 3
##   religion income      values
##   <chr>    <chr>      <dbl>
## 1 Agnostic <$10k         27
## 2 Agnostic $10-20k        34
## 3 Agnostic $20-30k        60
## 4 Agnostic $30-40k        81
## 5 Agnostic $40-50k        76
## 6 Agnostic $50-75k       137
## 7 Agnostic $75-100k      122
## 8 Agnostic $100-150k     109
## 9 Agnostic >150k         84
## 10 Agnostic Don't know/refused 96
## # ... with 170 more rows
```

Vale destacar, que você pode selecionar as colunas que você deseja transformar em linhas (argumento `cols`), através dos mesmos mecanismos que utilizamos na função `select()`. Ao eliminarmos a coluna `religion` com um sinal de menos (-) estávamos utilizando justamente um desses métodos. Mas podemos também, por exemplo, selecionar todas as colunas, que possuem dados de tipo numérico, com a função `is.numeric()`, atingindo o mesmo resultado anterior. Ou então, poderíamos selecionar todas as colunas que possuem em seu nome, algum dígito numérico, através da expressão regular "\\d" (*digit*) na função `matches()`.

```
relig_income %>%
  pivot_longer(
    cols = is.numeric,
    names_to = "income",
    values_to = "values"
  )

relig_income %>%
  pivot_longer(
    cols = matches("\\d"),
    names_to = "income",
    values_to = "values"
  )
```

Portanto, sempre que utilizar a função `pivot_longer()`, duas novas colunas serão criadas. Em uma dessas colunas (`values_to`), a função irá guardar os valores que se encontravam nas colunas que você transformou em linhas. Já na outra coluna (`names_to`), a função irá criar rótulos em cada linha, que lhe informam de qual coluna (que você transformou em linhas) veio o valor disposto na coluna anterior (`values_to`). Você sempre deve definir o nome dessas duas novas colunas, como texto, isto é, sempre forneça os nomes dessas colunas, entre aspas duplas ou simples.

Um outro exemplo, seria a tabela `billboard`, que também está disponível no pacote `tidyr`. Nessa tabela, temos a posição que diversas músicas ocuparam na lista da Billboard das 100 músicas mais populares no mundo, durante o ano de 2000. Portanto a posição que cada uma dessas músicas ocuparam nessa lista, ao longo do tempo, é a unidade básica que está sendo tratada nessa tabela. Agora, repare que a tabela possui muitas colunas (79 no total), onde em cada uma delas, temos a posição de uma música em uma dada semana desde a sua entrada na lista.

`billboard`

```
## # A tibble: 317 x 79
##   artist  track    date.entered  wk1  wk2  wk3  wk4  wk5  wk6  wk7
##   <chr>   <chr>    <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 2 Pac    Baby Do~ 2000-02-26    87   82   72   77   87   94   99
## 2 2Ge+her The Har~ 2000-09-02    91   87   92   NA   NA   NA   NA
## 3 3 Doors~ Krypton~ 2000-04-08    81   70   68   67   66   57   54
## 4 3 Doors~ Loser    2000-10-21    76   76   72   69   67   65   55
```

```
## 5 504 Boyz Wobble ~ 2000-04-15      57    34    25    17    17    31    36
## 6 98^0      Give Me~ 2000-08-19      51    39    34    26    26    19     2
## 7 A*Teens  Dancing~ 2000-07-08      97    97    96    95   100    NA    NA
## 8 Aaliyah  I Don't~ 2000-01-29      84    62    51    41    38    35    35
## 9 Aaliyah  Try Aga~ 2000-03-18      59    53    38    28    21    18    16
## 10 Adams, ~ Open My~ 2000-08-26      76    76    74    69    68    67    61
## # ... with 307 more rows, and 69 more variables: wk8 <dbl>, wk9 <dbl>,
## #   wk10 <dbl>, wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, wk14 <dbl>,
## #   wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>, wk19 <dbl>,
## #   wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
## #   wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>,
## #   wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,
## #   wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>,
## #   wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>,
## #   wk45 <dbl>, wk46 <dbl>, wk47 <dbl>, wk48 <dbl>, wk49 <dbl>,
## #   wk50 <dbl>, wk51 <dbl>, wk52 <dbl>, wk53 <dbl>, wk54 <dbl>,
## #   wk55 <dbl>, wk56 <dbl>, wk57 <dbl>, wk58 <dbl>, wk59 <dbl>,
## #   wk60 <dbl>, wk61 <dbl>, wk62 <dbl>, wk63 <dbl>, wk64 <dbl>,
## #   wk65 <dbl>, wk66 <lgl>, wk67 <lgl>, wk68 <lgl>, wk69 <lgl>,
## #   wk70 <lgl>, wk71 <lgl>, wk72 <lgl>, wk73 <lgl>, wk74 <lgl>,
## #   wk75 <lgl>, wk76 <lgl>
```

Repare também, que temos nessa tabela, mais semanas do que o total de semanas contidas em um ano corrido ($365/7 \approx 52$ semanas). Pela descrição das colunas restantes, que se encontra logo abaixo da tabela, vemos que a tabela possui dados até a 76ª semana (wk76). Isso provavelmente ocorre, porque algumas músicas que estão sendo descritas nessa tabela, entraram para a lista da Billboard no meio do ano anterior (1999), e portanto, permaneceram na lista mesmo durante o ano de 2000, ultrapassando o período de 1 ano, e portanto, de 52 semanas.

Agora, está claro que a forma como essa tabela está organizada, pode lhe trazer um trabalho imenso. Especialmente se você precisar aplicar uma função sobre cada uma dessas 76 colunas separadamente. Por isso, o ideal seria transformarmos todas essas 76 colunas, em novas linhas de sua tabela.

Porém, você não vai querer digitar o nome de cada uma dessas 76 colunas, no argumento `cols` de `pivot_longer()`. Novamente, quando há um conjunto muito grande de colunas que desejamos selecionar, podemos utilizar os métodos alternativos de seleção que vimos em `select()`. Por exemplo, podemos selecionar todas essas colunas pelo seus índices. No primeiro exemplo abaixo, estamos fazendo justamente isso, ao dizer à função em `cols`, que desejamos transformar todas as colunas entre a 4ª e a 79ª coluna. Uma outra alternativa, seria selecionarmos todas as colunas que possuem nomes que começam por “wk”, com a função `starts_with()`. Ambas alternativas, geram o mesmo resultado.

```
billboard_long <- billboard %>%
  pivot_longer(
    cols = 4:79,
```

```

    names_to = "week",
    values_to = "position"
  )

billboard_long <- billboard %>%
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "position"
  )

billboard_long

## # A tibble: 24,092 x 5
##   artist track          date.entered week position
##   <chr>  <chr>          <date>      <chr>    <dbl>
## 1 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk1      87
## 2 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk2      82
## 3 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk3      72
## 4 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk4      77
## 5 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk5      87
## 6 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk6      94
## 7 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk7      99
## 8 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk8      NA
## 9 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk9      NA
## 10 2 Pac   Baby Don't Cry (Keep... 2000-02-26 wk10     NA
## # ... with 24,082 more rows

```

Tais métodos de seleção são muito eficazes, e trazem grande otimização para o seu trabalho. Entretanto, em muitas ocasiões que utilizar essas funções de pivô, você vai precisar transformar apenas um conjunto pequeno de colunas em sua tabela. Nestes casos, talvez seja mais simples, definir diretamente os nomes das colunas que você deseja transformar, em cols. Veja por exemplo, a tabela df que eu crio logo abaixo.

```

df <- tibble(
  nome = c("Ana", "Eduardo", "Paulo"),
  `2005` = c(1800, 2100, 1230),
  `2006` = c(2120, 2100, 1450),
  `2007` = c(2120, 2100, 1980),
  `2008` = c(3840, 2100, 2430)
)

df

## # A tibble: 3 x 5
##   nome      `2005` `2006` `2007` `2008`

```

```
##   <chr>    <dbl> <dbl> <dbl> <dbl>
## 1 Ana      1800   2120   2120   3840
## 2 Eduardo  2100   2100   2100   2100
## 3 Paulo    1230   1450   1980   2430
```

Essa tabela contém os salários médios de três indivíduos hipotéticos, ao longo de quatro anos diferentes. Note que esses quatro anos, estão distribuídos ao longo de quatro colunas dessa tabela. Nesse exemplo, podemos utilizar novamente a função `pivot_longer()`, para transformarmos essas colunas em linhas. Dessa forma, temos o seguinte resultado:

```
df %>%
  pivot_longer(
    cols = c("2005", "2006", "2007", "2008"),
    names_to = "ano",
    values_to = "salario"
  )
```

```
## # A tibble: 12 x 3
##   nome   ano  salario
##   <chr> <chr>   <dbl>
## 1 Ana    2005    1800
## 2 Ana    2006    2120
## 3 Ana    2007    2120
## 4 Ana    2008    3840
## 5 Eduardo 2005    2100
## 6 Eduardo 2006    2100
## 7 Eduardo 2007    2100
## 8 Eduardo 2008    2100
## 9 Paulo   2005    1230
## 10 Paulo  2006    1450
## 11 Paulo  2007    1980
## 12 Paulo  2008    2430
```

7.3.2 Adicionando colunas à sua tabela com `pivot_wider()`

Por outro lado, você talvez deseje realizar a operação contrária. Ou seja, se você deseja transformar linhas de sua tabela, em novas colunas, você deve utilizar a função `pivot_wider()`, que possui argumentos muito parecidos com os de `pivot_longer()`.

Vamos começar com um exemplo simples. Veja a tabela `df` que estou criando logo abaixo. Nessa tabela, temos dados como o peso, a idade e a altura de cinco pessoas diferentes. Porém, perceba que essa tabela, não está no formato *tidy*. Pois temos três informações (peso, idade e altura) que representam características diferentes da unidade básica da tabela (pessoas), que estão em uma mesma coluna (variável).

```
df <- structure(list(nome = c("Ana", "Ana", "Ana", "Eduardo", "Eduardo",
"Eduardo", "Paulo", "Paulo", "Paulo", "Henrique", "Henrique",
"Henrique", "Letícia", "Letícia", "Letícia"), variavel = c("idade",
"peso", "altura", "idade", "peso", "altura", "idade", "peso",
"altura", "idade", "peso", "altura", "idade", "peso", "altura"
), valor = c(20, 61, 1.67, 18, 90, 1.89, 19, 68, 1.67, 23, 82,
1.72, 27, 56, 1.58)), row.names = c(NA, -15L), class = c("tbl_df",
"tbl", "data.frame"))
```

```
df
```

```
## # A tibble: 15 x 3
##   nome      variavel valor
##   <chr>    <chr>    <dbl>
## 1 Ana      idade      20
## 2 Ana      peso       61
## 3 Ana      altura    1.67
## 4 Eduardo idade      18
## 5 Eduardo peso       90
## 6 Eduardo altura    1.89
## 7 Paulo   idade      19
## 8 Paulo   peso       68
## 9 Paulo   altura    1.67
## 10 Henrique idade      23
## 11 Henrique peso       82
## 12 Henrique altura    1.72
## 13 Letícia idade      27
## 14 Letícia peso       56
## 15 Letícia altura    1.58
```

Portanto, tendo identificado o problema, precisamos agora, separar as três variáveis contidas na coluna `variavel`, em três novas colunas da tabela `df`. Logo, precisamos alargar a nossa base, pois estamos eliminando linhas e adicionando colunas à tabela.

Já sabemos que podemos utilizar a função `pivot_wider()` para esse trabalho, mas eu ainda não descrevi os seus argumentos, que são os seguintes: 1) `id_cols`, sendo as colunas que são suficientes para, ou capazes de, identificar uma única observação de sua base; 2) `names_from`, qual a coluna de sua tabela que contém as linhas a serem divididas, ou transformadas, em várias outras colunas; 3) `values_from`, qual a coluna, que contém os valores a serem posicionados nas novas células, que serão criadas durante o processo de “alargamento” da sua tabela.

Antes de prosseguirmos para os exemplos práticos, é provavelmente uma boa ideia, refletirmos sobre o que o argumento `id_cols` significa. Para que você identifique as colunas a serem estipuladas no argumento `id_cols`, você precisa primeiro identificar a unidade básica que está sendo tratada em sua tabela. No nosso caso, a tabela `df`, contém dados sobre características físicas ou biológicas, de cinco pessoas diferentes. Logo, a unidade básica dessa tabela, são as pessoas que estão sendo

descritas nela, e por isso, a coluna nome é capaz de identificar cada unidade básica, pois ela nos traz justamente um código social de identificação, isto é, o nome dessas pessoas.

Porém, repare que cada pessoa descrita na tabela `df`, não possui a sua própria linha na tabela. Veja por exemplo, as informações referentes à Ana, que estão definidas ao longo das três primeiras linhas da tabela. Com isso, eu quero apenas destacar que cada unidade básica, ou cada observação de sua tabela, não necessariamente vai se encontrar em um única linha, e que isso não deve ser uma regra (ou um guia) para selecionarmos as colunas de `id_cols`. Até porque, nós estamos utilizando uma operação de pivô sobre a nossa tabela, justamente pelo fato dela não estar no formato *tidy*. Ou seja, se uma das características que definem o formato *tidy*, não estão sendo respeitados, é muito provável, que cada observação de sua base, não se encontre em uma única linha.

Pensando em um outro exemplo, se você dispõe de uma base que descreve o PIB de cada município do estado de Minas Gerais, você precisa definir em `id_cols`, a coluna (ou o conjunto de colunas) que é capaz de identificar cada um dos 853 municípios de MG, pois esses municípios são a unidade básica da tabela. Porém, se a sua base está descrevendo o PIB desses mesmos municípios, mas agora ao longo dos anos de 2010 a 2020, a sua unidade básica passa a ter um componente temporal, e se torna a evolução desses municípios ao longo do tempo. Dessa forma, você precisaria não apenas de uma coluna que seja capaz de identificar qual o município que está sendo descrito na base, mas também de uma outra coluna que possa identificar qual o ano que a informação desse município se refere.

Tendo isso em mente, vamos partir para os próximos dois argumentos. No nosso caso, queremos pegar as três variáveis que estão ao longo da coluna `variavel`, e separá-las em três colunas diferentes. Isso é exatamente o que devemos definir em `names_from`. O que este argumento está pedindo, é o nome da coluna que contém os valores que vão servir de nome para as novas colunas que `pivot_wider()` irá criar. Ou seja, ao fornecermos a coluna `variavel` para `names_from`, `pivot_wider()` irá criar uma nova coluna para cada valor único que se encontra na coluna `variavel`. Como ao longo da coluna `variavel`, temos três valores diferentes (peso, altura e idade), `pivot_wider()` irá criar três novas colunas que possuem os nomes de peso, altura e idade.

Ao criar as novas colunas, você precisa preenchê-las de alguma forma, a menos que você deseje deixá-las vazias. Em outras palavras, a função `pivot_wider()` irá lhe perguntar: “Ok, eu criei as colunas que você me pediu para criar, mas eu devo preenchê-las com que valores?”. Você deve responder essa pergunta, através do argumento `values_from`, onde você irá definir qual é a coluna que contém os valores que você deseja alocar ao longo dessas novas colunas (que foram criadas de acordo com os valores contidos na coluna que você definiu em `names_from`). Na nossa tabela `df`, é a coluna `valor` que contém os registros, ou os valores que cada variável (idade, altura e peso) assume nessa amostra. Logo, é essa coluna que devemos conectar à `values_from`.

```
df %>%  
  pivot_wider(  
    id_cols = nome,  
    names_from = variavel,  
    values_from = valor
```

```
)

## # A tibble: 5 x 4
##   nome      idade  peso altura
##   <chr>    <dbl> <dbl>  <dbl>
## 1 Ana         20    61   1.67
## 2 Eduardo     18    90   1.89
## 3 Paulo       19    68   1.67
## 4 Henrique    23    82   1.72
## 5 Letícia     27    56   1.58
```

Esse foi um exemplo simples de como utilizar a função, e que vai lhe servir de base para praticamente qualquer aplicação de `pivot_wider()`. Porém, em algumas situações que você utilizar `pivot_wider()`, pode ser que a sua tabela não possua colunas o suficiente, que possam identificar unicamente cada observação de sua base, e isso, ficará mais claro com um outro exemplo.

Com o código abaixo, você é capaz de recriar a tabela vendas, em seu R. Lembre-se de executar a função `set.seed()` antes de criar a tabela vendas, pois é essa função que garante que você irá recriar exatamente a mesma tabela que a minha. Nessa tabela vendas, possuímos vendas hipótéticas de diversos produtos (identificados por `produtoid`), realizadas por alguns vendedores (identificados por `usuario`) que arrecadaram em cada venda os valores descritos na coluna `valor`. Perceba também, que essas vendas são diárias, pois possuímos outras três colunas (`ano`, `mes` e `dia`) que definem o dia em que a venda ocorreu.

```
nomes <- c("Ana", "Eduardo", "Paulo", "Henrique", "Letícia")
produto <- c("10032", "10013", "10104", "10555", "10901")
```

```
set.seed(1)
vendas <- tibble(
  ano = sample(2010:2020, size = 10000, replace = TRUE),
  mes = sample(1:12, size = 10000, replace = TRUE),
  dia = sample(1:31, size = 10000, replace = TRUE),
  usuario = sample(nomes, size = 10000, replace = TRUE),
  valor = rnorm(10000, mean = 5000, sd = 1600),
  produtoid = sample(produto, size = 10000, replace = TRUE)
) %>%
  arrange(ano, mes, dia, usuario)
```

vendas

```
## # A tibble: 10,000 x 6
##   ano  mes  dia usuario  valor produtoid
##   <int> <int> <int> <chr>    <dbl> <chr>
## 1  2010     1     1 Ana      3907. 10104
## 2  2010     1     1 Henrique 6139. 10104
```



```
## 3 2010 1 2 Henrique 5510. 10013
## 4 2010 1 3 Ana 5296. 10555
## 5 2010 1 3 Letícia 3525. 10555
## 6 2010 1 4 Ana 5102. 10555
## 7 2010 1 4 Eduardo 6051. 10013
## 8 2010 1 4 Letícia 4600. 10032
## 9 2010 1 5 Paulo 5869. 10104
## 10 2010 1 6 Ana 7188. 10013
## # ... with 9,990 more rows
```

Vou antes de mais nada, identificar os níveis (ou valores únicos) contidos nas duas colunas que vão servir de objeto de estudo, para os próximos exemplos. Caso você queira visualizar todos os valores únicos contidos em uma coluna, você pode realizar tal ação através da função `unique()`. Perceba pelos resultados abaixo, que nós temos cinco vendedores e cinco produtos diferentes que estão sendo descritos ao longo da tabela vendas.

```
unique(vendas$usuario)
```

```
## [1] "Ana" "Henrique" "Letícia" "Eduardo" "Paulo"
```

```
unique(vendas$produtoid)
```

```
## [1] "10104" "10013" "10555" "10032" "10901"
```

Portanto, vamos para o exemplo. Já adianto, que se você tentar distribuir tanto os vendedores (usuario), quanto os produtos vendidos (produtoid), em novas colunas de nossa tabela, utilizando `pivot_wider()`, um aviso será levantado, e o resultado dessa operação (apesar de correto) será provavelmente, muito estranho para você. Primeiro, veja com os seus próprios olhos, qual é o resultado dessa aplicação, com a coluna usuario:

```
vendas_wide <- vendas %>%
  pivot_wider(
    id_cols = c("ano", "mes", "dia", "produtoid"),
    names_from = usuario,
    values_from = valor
  )
```

```
## Warning: Values are not uniquely identified; output will contain list-cols.
## * Use `values_fn = list` to suppress this warning.
## * Use `values_fn = length` to identify where the duplicates arise
## * Use `values_fn = {summary_fun}` to summarise duplicates
```

```
vendas_wide
```

```
## # A tibble: 7,845 x 9
##   ano   mes   dia produtoid Ana      Henrique  Leticia  Eduardo  Paulo
##   <int> <int> <int> <chr>    <list>   <list>    <list>   <list>   <list>
## 1  2010     1     1 10104    <dbl [1~ <dbl [1]> <NULL>   <NULL>   <NULL>
## 2  2010     1     2 10013    <NULL>   <dbl [1]> <NULL>   <NULL>   <NULL>
## 3  2010     1     3 10555    <dbl [1~ <NULL>    <dbl [1~ <NULL>   <NULL>
## 4  2010     1     4 10555    <dbl [1~ <NULL>    <NULL>   <NULL>   <NULL>
## 5  2010     1     4 10013    <NULL>   <NULL>    <NULL>   <dbl [1~ <NULL>
## 6  2010     1     4 10032    <NULL>   <NULL>    <dbl [1~ <NULL>   <NULL>
## 7  2010     1     5 10104    <NULL>   <NULL>    <NULL>   <NULL>   <dbl [~
## 8  2010     1     6 10013    <dbl [1~ <NULL>    <NULL>   <NULL>   <NULL>
## 9  2010     1     6 10901    <NULL>   <NULL>    <dbl [1~ <NULL>   <NULL>
## 10 2010     1     7 10013    <NULL>   <dbl [1]> <NULL>   <NULL>   <dbl [~
## # ... with 7,835 more rows
```

Como podemos ver pelo resultado acima, uma mensagem de aviso apareceu, nos informando que os valores não podem ser unicamente identificados através das colunas que fornecemos em `id_cols` (*Values are not uniquely identified*), e que por isso, a função `pivot_wider()`, acabou transformando as novas colunas que criamos, em listas (*output will contain list-cols.*).

Ou seja, cada uma das colunas que acabamos de criar com `pivot_wider()`, estão na estrutura de um vetor recursivo (i.e. listas). Isso pode ser estranho para muitos usuários, pois na maioria das vezes, as colunas de suas tabelas serão vetores atômicos². Um outro motivo que provavelmente levantou bastante dúvida em sua cabeça é: “Como assim as colunas que forneci não são capazes de identificar unicamente os valores? Em que sentido elas não são capazes de realizar tal ação?”. Bem, essa questão ficará mais clara, se nos questionarmos como, ou por que motivo essas colunas foram transformadas para listas.

Antes de continuarmos, vale ressaltar que as novas colunas criadas por `pivot_wider()` nunca chegaram a ser colunas comuns, formadas por vetores atômicos. Logo, desde a sua criação, elas já eram listas. Mas se partirmos do pressuposto que inicialmente, essas colunas eram vetores atômicos, tal pensamento se torna útil para identificarmos os motivos para o uso de listas. Estes motivos serão identificados a seguir.

Primeiro, precisamos transformar novamente essas colunas em vetores atômicos, para tentarmos compreender como essas colunas ficariam como simples vetores atômicos. Para isso, vou pegar um pedaço da tabela `vendas`, mais especificamente, as 10 primeiras linhas da tabela, através da função `head()`. Em seguida, vou me preocupar em transformar essas colunas novamente em vetores, através dos comandos abaixo.

```
pedaco <- head(vendas_wide, 10)
```

²Apesar de serem um caso raro no R, as tabelas que possuem listas como colunas, tem se tornado cada vez mais comuns ao longo de diversas análises, e são comumente chamadas pela comunidade de *nested tables*, ou de *nested data*. Alguns pacotes tem se desenvolvido, de maneira muito forte nessa área, e por isso, essas estruturas tem se tornado de grande utilidade em diversas aplicações. Alguns desses pacotes incluem o próprio `tidyr`, além do pacote `broom`.

```
for(i in 5:9){
  id <- vapply(pedaco[[i]], FUN = is.null, FUN.VALUE = TRUE)
  pedaco[[i]][id] <- NA_real_
}

pedaco <- pedaco %>% mutate(across(5:9, unlist))
```

Após executarmos as transformações acima, possuímos agora, uma tabela comum, como qualquer outra que você encontra normalmente no R. Veja o resultado abaixo, quando chamamos pelo nome da tabela no console. Dessa vez, nas células que possuíam uma lista nula <NULL>(uma lista vazia) temos um valor de NA (não disponível). Já nas células que possuíam uma lista com algum valor, vemos agora, o valor exato que estava contido nessa lista, ao invés da descrição <dbl [1]>.

```
pedaco
```

```
## # A tibble: 10 x 9
##   ano   mes   dia produtoid   Ana Henrique Letícia Eduardo Paulo
##   <int> <int> <int> <chr>         <dbl>    <dbl>    <dbl>    <dbl> <dbl>
## 1  2010     1     1  10104      3907.    6139.     NA       NA    NA
## 2  2010     1     2  10013         NA    5510.     NA       NA    NA
## 3  2010     1     3  10555      5296.     NA    3525.     NA    NA
## 4  2010     1     4  10555      5102.     NA       NA       NA    NA
## 5  2010     1     4  10013         NA       NA       NA    6051.    NA
## 6  2010     1     4  10032         NA       NA    4600.     NA    NA
## 7  2010     1     5  10104         NA       NA       NA       NA  5869.
## 8  2010     1     6  10013      7188.     NA       NA       NA    NA
## 9  2010     1     6  10901         NA       NA    4491.     NA    NA
## 10 2010     1     7  10013         NA    4407.     NA       NA  2292.
```

Portanto, se observarmos a primeira linha dessa nova tabela pedaco, vemos que a vendedora Ana, vendeu no dia 01/01/2010, o produto de ID 10104, no valor de 3907 reais (e alguns centavos). Neste mesmo dia, o Henrique vendeu o mesmo produto, por quase 2 mil reais a mais que Ana, totalizando 6139 reais de receita. Podemos perceber também pelas outras colunas, que nenhum outro vendedor conseguiu vender uma unidade do produto de ID 10104, no dia 01/01/2010.

Neste ponto, se pergunte: “Ok, Ana vendeu uma unidade do produto 10104, no dia 01. Mas e se ela tivesse vendido duas unidades desse mesmo produto 10104, no dia 01?”. Tente imaginar, como os dados dessas duas vendas ficariam na tabela. Isso não é uma questão trivial, pois temos agora dados de duas vendas diferentes...mas apenas uma célula disponível em nossa tabela, para guardar esses dados. É a partir deste choque, que podemos identificar qual foi o motivo para o uso de listas nas novas colunas.

Dito de outra forma, temos uma única célula na tabela (localizada na primeira linha, e quinta coluna da tabela), que deve conter o valor arrecadado na venda do produto 10104, realizada pela vendedora Ana no dia 01/01/2010. Você poderia pensar: “Bem, por que não somar os valores dessas duas vendas? Dessa forma, temos apenas um valor para encaixar nessa célula”. Essa é uma alternativa possível, porém, ela gera perda de informação, especialmente se o valor arrecadado nas duas operações forem diferentes. Por exemplo, se a receita da primeira venda foi de 3907, e da segunda, de 4530. Com essa alternativa, nós sabemos que a soma das duas vendas ocorridas naquele dia, geraram 8437 reais de receita, mas nós não sabemos mais, qual foi o menor valor arrecadado nas duas operações.

Isso é particularmente importante, pois podemos gerar o mesmo valor (8437 reais) de múltiplas formas. Pode ser que a Ana tenha vendido cinco unidades do produto 10104, tendo arrecadado em cada venda, o valor de 1687,4 reais. Mas ela poderia atingir o mesmo valor, ao vender dez unidades do produto 10104, dessa vez, arrecadando um valor médio bem menor, de 843,7 reais. Portanto, se utilizarmos a soma desses valores, como forma de contornarmos o problema posto anteriormente, os administradores da loja não poderão mais inferir da tabela vendas, se as suas vendas tem se reduzido em quantidade, ou se o valor arrecadado em cada venda, tem caído ao longo dos últimos anos.

Apesar de ser uma alternativa ruim para muitos casos, pode ser desejável agregar as informações dessas vendas em uma só para o seu caso. Nesta situação, as versões mais recentes do pacote `tidyr`, oferecem na função `pivot_wider()` o argumento `values_fn`, onde você pode fornecer o nome de uma função a ser aplicada sobre os valores dispostos em cada célula. Logo, se quiséssemos somar os valores de vendas dispostos em cada célula criadas na tabela `vendas_wide`, poderíamos realizar os comandos abaixo:

```
vendas %>%
  pivot_wider(
    id_cols = c("ano", "mes", "dia", "produtoid"),
    names_from = usuario,
    values_from = valor,
    values_fn = sum
  )
```

A tibble: 7,845 x 9

	ano	mes	dia	produtoid	Ana	Henrique	Letícia	Eduardo	Paulo
	<int>	<int>	<int>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
## 1	2010	1	1	10104	3907.	6139.	NA	NA	NA
## 2	2010	1	2	10013	NA	5510.	NA	NA	NA
## 3	2010	1	3	10555	5296.	NA	3525.	NA	NA
## 4	2010	1	4	10555	5102.	NA	NA	NA	NA
## 5	2010	1	4	10013	NA	NA	NA	6051.	NA
## 6	2010	1	4	10032	NA	NA	4600.	NA	NA
## 7	2010	1	5	10104	NA	NA	NA	NA	5869.
## 8	2010	1	6	10013	7188.	NA	NA	NA	NA

```
## 9 2010 1 6 10901 NA NA 4491. NA NA
## 10 2010 1 7 10013 NA 4407. NA NA 2292.
## # ... with 7,835 more rows
```

que se você tentar adicionar a coluna valor a `id_cols`, você irá perceber que nada se altera, e o mesmo resultado é gerado.

Portanto, não há uma resposta fácil para uma situação como essa, onde mesmo fornecendo todas as colunas para `id_cols` em `pivot_wider()`, a função ainda não é capaz de identificar unicamente cada valor da coluna que você forneceu em `value_from`. Você pode utilizar uma solução que gera perda de informação, ao aplicar uma função sumária, ou seja, uma função para agregar esses valores de forma que eles se tornem únicos, dados os conjuntos de colunas que você forneceu em `id_cols`. Uma outra possibilidade, é que você esteja utilizando a operação de pivô errada. Ou seja, a melhor alternativa seria alongar (`pivot_longer()`) a sua base, ao invés de alargá-la.

Agora, uma última possibilidade mais promissora, é que você esteja realizando a operação correta, e que faz sentido manter essas colunas como listas de acordo com o que você deseja realizar com a base. Isso inclui o uso de um ferramental que está um pouco além desse capítulo. Por outro lado, lidar com *nested data*, é mais uma questão de experiência, de se acostumar com tal estrutura, e saber as funções adequadas, do que aprender algo muito diferente do que mostramos aqui. Um outro conhecimento que é de extrema importância nessas situações, é conhecer muito bem como as listas funcionam no R. Se você conhecer bem essa estrutura, você não terá dificuldades em navegar por *nested data*. Para uma visão melhor do potencial que *nested data* pode trazer para sua análise, eu recomendo que você procure por uma excelente palestra de Hadley Wickham, intitulada “*Managing many models with R*”³.

7.4 Completando e expandindo a sua tabela

A operação que vou mostrar a seguir, serve para completar, ou inserir linhas que estão faltando em sua tabela. Em outras palavras, essa operação busca tornar os valores que estão implicitamente faltando em sua tabela, em valores não disponíveis explícitos. Você também pode enxergar esse processo, como uma forma rápida de expandir a sua tabela, a partir de combinações de valores. Um exemplo lógico do uso dessa operação, seriam datas que você gostaria que estivessem em sua tabela, mas que não se encontram nela no momento. Vamos supor por exemplo, que você possua a tabela abaixo:

```
library(tidyverse)

dias <- c("2020-09-01", "2020-09-05", "2020-09-07", "2020-09-10")

set.seed(1)
vendas <- tibble(
  datas = as.Date(dias),
  nome = c("Ana", "Julia", "Joao", "Julia"),
  valor = rnorm(4, mean = 500, sd = 150)
)
```

³https://www.youtube.com/watch?v=rz3_FDvt9eg&ab_channel=PsychologyattheUniversityofEdinburgh

vendas

```
## # A tibble: 4 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
## 1 2020-09-01 Ana    406.
## 2 2020-09-05 Julia  528.
## 3 2020-09-07 Joao   375.
## 4 2020-09-10 Julia  739.
```

Portanto, temos nessa tabela vendas, o nome de alguns vendedores e os valores de suas vendas efetuadas em alguns dias diferentes. No momento, temos vendas explícitas apenas nos dias 01, 05, 07 e 10 de Setembro de 2020, mas o que ocorreu nos dias que estão entre essas datas (dias 02, 03, 04, 06, 08 e 09 de Setembro de 2020)? Caso você estivesse apresentando esses dados para o seu chefe, por exemplo, essa seria uma questão que ele provavelmente faria a você.

Bem, vamos supor que não tenham ocorrido vendas durante esses dias, e que por isso eles não estão sendo descritos na tabela vendas. Talvez seja de seu desejo, introduzir esses dias na tabela para que ninguém fique em dúvida a respeito desses dias. Com isso, precisamos então completar a tabela vendas, com linhas que estão implicitamente faltando nela.

7.4.1 Encontrando possíveis combinações com a função `expand()`

Apesar não ser exatamente o que desejamos para a tabela vendas, o processo em que buscamos encontrar possíveis combinações de dados que não estão presentes em nossa tabela, também envolve a procura por todas as combinações possíveis dos dados presentes nessa tabela. Nessa seção, vamos introduzir alguns métodos para encontrarmos todas as combinações possíveis de seus dados.

Para isso, podemos utilizar a função `expand()` do pacote `tidyr`. Essa função busca expandir uma tabela, de forma que ela inclua todas as possíveis combinações de certos valores. Em maiores detalhes, essa função irá criar (com base nos dados que você fornecer a ela) uma nova tabela, ou um novo tibble, que irá incluir todas as combinações únicas e possíveis dos valores que você definiu. Portanto, se eu fornecer a tabela vendas à função, e pedir a ela que encontre todas as combinações possíveis entre os valores contidos nas colunas datas e nomes, esse será o resultado:

```
expand(vendas, datas, nome)
```

```
## # A tibble: 12 x 2
##   datas      nome
##   <date>    <chr>
## 1 2020-09-01 Ana
## 2 2020-09-01 Joao
## 3 2020-09-01 Julia
## 4 2020-09-05 Ana
```

```
## 5 2020-09-05 Joao
## 6 2020-09-05 Julia
## 7 2020-09-07 Ana
## 8 2020-09-07 Joao
## 9 2020-09-07 Julia
## 10 2020-09-10 Ana
## 11 2020-09-10 Joao
## 12 2020-09-10 Julia
```

Portanto, `expand()` irá criar uma nova tabela, contendo todas as possíveis combinações entre os valores das colunas datas e nomes da tabela vendas. Incluindo aquelas combinações que não aparecem na tabela inicial. Por exemplo, as combinações (2020-09-01, Julia), ou (2020-09-05, Joao) e (2020-09-10, Joao) não estão presentes na tabela vendas, e mesmo assim foram introduzidas no resultado de `expand()`.

Porém, `expand()` não definiu novas combinações com as datas que estão faltando na tabela vendas (por exemplo, os dias 02, 03, 04 e 08 de Setembro de 2020). Ou seja, em nenhum momento `expand()` irá adicionar algum dado à sua tabela, seja antes ou depois de encontrar todas as combinações únicas. Em outras palavras, `expand()` irá sempre encontrar todas as combinações possíveis, se baseando nos valores que já se encontram nas variáveis que você forneceu a ela. Por isso, mesmo que a combinação (2020-09-01, Julia) não esteja definida na tabela vendas, ela é uma combinação possível, pois os valores 2020-09-01 e Julia estão presentes na tabela vendas.

Vale destacar, que você pode combinar as variáveis de sua tabela, com vetores externos. Por exemplo, eu posso utilizar `seq.Date()` para gerar todas as datas que estão entre o dia 01 e 10 de Setembro de 2020. No exemplo abaixo, perceba que `expand()` pega cada um dos 3 nomes únicos definidos na coluna nome de vendas, e combina eles com cada uma das 10 datas guardadas no vetor `vec_d`, gerando assim, uma nova tabela com 30 linhas (3 nomes $\times$ 10 datas = 30 combinações).

```
vec_d <- seq.Date(min(vendas$datas), max(vendas$datas), by = "day")
```

```
expand(vendas, nome, vec_d)
```

```
## # A tibble: 30 x 2
##   nome  vec_d
##   <chr> <date>
## 1 Ana   2020-09-01
## 2 Ana   2020-09-02
## 3 Ana   2020-09-03
## 4 Ana   2020-09-04
## 5 Ana   2020-09-05
## 6 Ana   2020-09-06
## 7 Ana   2020-09-07
## 8 Ana   2020-09-08
## 9 Ana   2020-09-09
```



```
## 10 Ana    2020-09-10
## # ... with 20 more rows
```

Além disso, a função `expand()` conta com uma função auxiliar útil (`nesting()`), que restringe quais combinações serão válidas para `expand()`. Ao incluir variáveis dentro da função `nesting()`, você está dizendo à `expand()`, que encontre apenas as combinações únicas (entre os valores dessas variáveis) que já estão presentes em sua base. Ou seja, se eu colocar as colunas `datas` e `nome` dentro de `nesting()`, a função `expand()` irá basicamente repetir a tabela `vendas`. Pois cada uma das 4 linhas (ou 4 combinações entre `datas` e `nome`), aparecem uma única vez nessa tabela.

```
expand(vendas, nesting(datas, nome))
```

```
## # A tibble: 4 x 2
##   datas      nome
##   <date>    <chr>
## 1 2020-09-01 Ana
## 2 2020-09-05 Julia
## 3 2020-09-07 Joao
## 4 2020-09-10 Julia
```

Dessa maneira, o uso de `nesting()` acima, é análogo ao uso da função `unique()` que vêm dos pacotes básicos do R. Logo, poderíamos atingir exatamente o mesmo resultado, utilizando qualquer uma das duas funções. Podemos por exemplo, adicionarmos uma quinta linha à tabela `vendas`, que repete os valores contidos na quarta linha da tabela. Perceba abaixo, que ao utilizarmos `unique()` ou `nesting()`, em ambos os casos, essa quinta linha repetida desaparece. Pois ambas as funções buscam encontrar todas as combinações **únicas** que aparecem ao longo da tabela `vendas`.

```
vendas[5, ] <- data.frame(as.Date("2020-09-10"), "Julia", 739.29)
```

```
vendas
```

```
## # A tibble: 5 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
## 1 2020-09-01 Ana    406.
## 2 2020-09-05 Julia  528.
## 3 2020-09-07 Joao   375.
## 4 2020-09-10 Julia  739.
## 5 2020-09-10 Julia  739.
```

```
# Estou aplicando unique() sobre a
# primeira e segunda coluna de vendas
unique(vendas[, 1:2])
```

```
## # A tibble: 4 x 2
##   datas      nome
##   <date>    <chr>
## 1 2020-09-01 Ana
## 2 2020-09-05 Julia
## 3 2020-09-07 Joao
## 4 2020-09-10 Julia

# O mesmo resultado pode ser
# atingido com o uso de nesting() em expand()
expand(vendas, nesting(datas, nome))

## # A tibble: 4 x 2
##   datas      nome
##   <date>    <chr>
## 1 2020-09-01 Ana
## 2 2020-09-05 Julia
## 3 2020-09-07 Joao
## 4 2020-09-10 Julia
```

Vale destacar que você pode combinar o comportamento restrito e irrestrito de `expand()`. Ou seja, você pode restringir as combinações com o uso de `nesting()` para algumas variáveis, enquanto outras permanecem de fora dessa função, permitindo uma gama maior de combinações. No exemplo abaixo, `expand()` vai encontrar primeiro, cada combinação única entre nome e valor que está presente na tabela vendas, em seguida, a função irá encontrar **todas** as combinações possíveis entre as combinações anteriores (entre nome e valor) e todas as datas descritas na base.

Em outras palavras, nós podemos encontrar no resultado abaixo, uma combinação como (2020-09-01, Julia, 528.). Pois a combinação (Julia, 528.) existe nas colunas nome e valor da tabela vendas, e como deixamos a coluna datas de fora de `nesting()`, `expand()` irá combinar (Julia, 528.) com toda e qualquer data disponível na tabela vendas.

Porém, nós não podemos encontrar no resultado abaixo, uma combinação como (2020-09-01, Ana, 739.). Pois a única combinação entre as colunas nome e valor, presente na tabela vendas, que possui o valor 739 na coluna valor, é a linha que contém a combinação (Julia, 739.). Logo, se não há nas colunas nome e valor alguma combinação entre Ana e o valor 739., `expand()` não irá combinar esses valores com todas as datas disponíveis na base. Pois as combinações entre as colunas nome e valor estão sendo restringidas por `nesting()`.

```
vendas %>%
  expand(datas, nesting(nome, valor))

## # A tibble: 16 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
```

```
## 1 2020-09-01 Ana 406.
## 2 2020-09-01 Joao 375.
## 3 2020-09-01 Julia 528.
## 4 2020-09-01 Julia 739.
## 5 2020-09-05 Ana 406.
## 6 2020-09-05 Joao 375.
## 7 2020-09-05 Julia 528.
## 8 2020-09-05 Julia 739.
## 9 2020-09-07 Ana 406.
## 10 2020-09-07 Joao 375.
## 11 2020-09-07 Julia 528.
## 12 2020-09-07 Julia 739.
## 13 2020-09-10 Ana 406.
## 14 2020-09-10 Joao 375.
## 15 2020-09-10 Julia 528.
## 16 2020-09-10 Julia 739.
```

7.4.2 A metodologia por detrás do processo

Apesar de próximo, a função `expand()` não é suficiente para produzirmos o resultado que desejamos. Lembre-se que nós temos a tabela abaixo, e que desejamos completá-la com os dados referentes aos dias 02, 03, 04, 06, 08 e 09 de Setembro de 2020, que estão no momento faltando nessa tabela.

vendas

```
## # A tibble: 4 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
## 1 2020-09-01 Ana    406.
## 2 2020-09-05 Julia  528.
## 3 2020-09-07 Joao   375.
## 4 2020-09-10 Julia  739.
```

Primeiro, precisamos encontrar todos valores possíveis da variável que está incompleta na tabela vendas. Ou seja, queremos encontrar todas as datas possíveis entre os dias 01 e 10 de Setembro de 2020, pois esses dias são os limites da tabela. Dito de outra forma, a tabela vendas descreve dados de vendas que ocorreram do dia 01 até o dia 10 de Setembro de 2020. Por isso, queremos encontrar todos os dias possíveis entre esse intervalo de tempo.

Para isso, podemos utilizar a função `seq.Date()` em conjunto com `tibble()`. Dessa forma, nós criamos uma nova tabela, que contém uma sequência de datas que vai do dia 01 até o dia 10 de Setembro. O mesmo resultado, poderia ser atingido, caso utilizássemos `seq.Date()` dentro de `expand()`, já que `expand()` cria por padrão uma nova tabela com todas as combinações possíveis dos dados que você fornece a ela.

```
nova_tab <- tibble(
  datas = seq.Date(min(vendas$datas), max(vendas$datas), by = "day")
)
```

```
nova_tab
```

```
## # A tibble: 10 x 1
##   datas
##   <date>
## 1 2020-09-01
## 2 2020-09-02
## 3 2020-09-03
## 4 2020-09-04
## 5 2020-09-05
## 6 2020-09-06
## 7 2020-09-07
## 8 2020-09-08
## 9 2020-09-09
## 10 2020-09-10
```

O mesmo resultado poderia ser atingido com:

```
nova_tab <- expand(
  datas = seq.Date(min(vendas$datas), max(vendas$datas), by = "day")
)
```

Em seguida, podemos utilizar a função `full_join()`⁴ do pacote `dplyr`, para trazermos os dados disponíveis na tabela `vendas` para essa nova tabela `nova_tab`. Agora, nós temos uma nova tabela, que contém todos os dados que já estão definidos na tabela `vendas`, além dos dias que estavam faltando anteriormente, e que agora também estão definidos.

```
nova_tab <- nova_tab %>%
  full_join(vendas, by = "datas")
```

```
nova_tab
```

```
## # A tibble: 10 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
## 1 2020-09-01 Ana    406.
## 2 2020-09-02 <NA>    NA
## 3 2020-09-03 <NA>    NA
```

⁴Caso você não conheça a função `full_join()`, lembre-se que ela é descrita em detalhes no capítulo intitulado “Introdução a base de dados relacionais”.

```
## 4 2020-09-04 <NA>    NA
## 5 2020-09-05 Julia  528.
## 6 2020-09-06 <NA>    NA
## 7 2020-09-07 Joao   375.
## 8 2020-09-08 <NA>    NA
## 9 2020-09-09 <NA>    NA
## 10 2020-09-10 Julia  739.
```

O que resta agora, é preencher os campos com valores não-disponíveis (NA) com algum outro valor que seja mais claro, ou que indique de um forma melhor, que não houve vendas realizadas naquele dia. Visando esse objetivo, temos a função `replace_na()` do pacote `tidyr`. Nessa função, você irá fornecer uma lista (`list()`) contendo os valores que vão substituir os valores NA em cada coluna de sua tabela. Essa lista precisa ser nomeada. Basta nomear cada valor substituto com o nome da coluna em que você deseja utilizar esse valor. Logo, se eu quero substituir todos os valores NA na coluna `valor`, por um zero, basta eu nomear esse zero com o nome dessa coluna, dentro da lista (`list()`) que eu forneci à `replace_na()`.

```
nova_tab %>%
  replace_na(
    list(nome = "Não houve vendas", valor = 0)
  )
```

```
## # A tibble: 10 x 3
##   datas      nome      valor
##   <date>    <chr>    <dbl>
## 1 2020-09-01 Ana      406.
## 2 2020-09-02 Não houve vendas  0
## 3 2020-09-03 Não houve vendas  0
## 4 2020-09-04 Não houve vendas  0
## 5 2020-09-05 Julia    528.
## 6 2020-09-06 Não houve vendas  0
## 7 2020-09-07 Joao     375.
## 8 2020-09-08 Não houve vendas  0
## 9 2020-09-09 Não houve vendas  0
## 10 2020-09-10 Julia    739.
```

7.4.3 A função `complete()` como um atalho útil

A função `complete()` é um *wrapper*, ou uma função auxiliar do pacote `tidyr`, que engloba as funções `expand()`, `full_join()`, e `replace_na()`. Ou seja, a função `complete()` é um atalho para aplicarmos a metodologia que acabamos de descrever na seção anterior. A função possui três argumentos principais: 1) `data`, o nome do objeto onde a sua tabela está salva; 2) `...`, a especificação das colunas a serem completadas, ou “expandidas” por `complete()`; 3) `fill`, uma lista nomeada (como a que fornecemos em `replace_na()`), que atribui para cada variável (ou coluna) de sua tabela, um valor a ser utilizado (ao invés de NA) para as combinações faltantes.

Vou explicar o argumento `fill` mais a frente, por isso, vamos nos concentrar nos outros dois. A tabela que contém os nossos dados se chama `vendas`, e por isso, é esse valor que devemos atribuir ao argumento `data`. Porém, como estamos utilizando o *pipe* (`%>%`) no exemplo abaixo, ele já está realizando esse serviço para nós. Já o segundo argumento (`...`), diz respeito a lista de especificações que vão definir como a função `complete()` deve completar cada coluna da nossa tabela.

Em outras palavras, o segundo argumento (`...`) é a parte da função `complete()` que diz respeito ao uso de `expand()`. Você deve portanto, preencher este argumento, da mesma forma que você faria com a função `expand()`. No exemplo abaixo, o primeiro argumento (`data`), já está sendo definido pelo operador *pipe* (`%>%`). Perceba que eu preencho a função `complete()`, da mesma forma em que preenchi a função `expand()` na seção anterior. Perceba também, que `complete()` já me retorna como resultado, a tabela expandida após o uso de `full_join()`.

```
vendas %>%
  complete(
    datas = seq.Date(min(datas), max(datas), by = "day")
  )
```

```
## # A tibble: 10 x 3
##   datas      nome  valor
##   <date>    <chr> <dbl>
## 1 2020-09-01 Ana    406.
## 2 2020-09-02 <NA>    NA
## 3 2020-09-03 <NA>    NA
## 4 2020-09-04 <NA>    NA
## 5 2020-09-05 Julia  528.
## 6 2020-09-06 <NA>    NA
## 7 2020-09-07 Joao   375.
## 8 2020-09-08 <NA>    NA
## 9 2020-09-09 <NA>    NA
## 10 2020-09-10 Julia  739.
```

O último passo que resta agora, seria o uso de `replace_na()` para preencher os valores não-disponíveis por algum outro valor mais claro. Nós ainda podemos utilizar a função `complete()` para executarmos esse passo. Basta você fornecer à `complete()` através de seu terceiro argumento (`fill`), a mesma lista que você forneceria à `replace_na()`. Dessa forma, temos:

```
vendas %>%
  complete(
    datas = seq.Date(min(datas), max(datas), by = "day"),
    fill = list(nome = "Não houve vendas", valor = 0)
  )
```

```
## # A tibble: 10 x 3
##   datas      nome      valor
```

```
##      <date>      <chr>          <dbl>
## 1 2020-09-01 Ana              406.
## 2 2020-09-02 Não houve vendas    0
## 3 2020-09-03 Não houve vendas    0
## 4 2020-09-04 Não houve vendas    0
## 5 2020-09-05 Julia            528.
## 6 2020-09-06 Não houve vendas    0
## 7 2020-09-07 Joao             375.
## 8 2020-09-08 Não houve vendas    0
## 9 2020-09-09 Não houve vendas    0
## 10 2020-09-10 Julia           739.
```

7.5 Preenchendo valores não-disponíveis (NA)

7.5.1 Utilizando-se de valores anteriores ou posteriores

As operações que vou mostrar a seguir, servem para preencher linhas com dados não-disponíveis (NA), com valores anteriores ou posteriores que estão disponíveis em sua tabela. Vamos começar com um exemplo simples através da tabela `df`, que você pode criar em seu R utilizando os comandos abaixo. Nessa tabela, temos algumas vendas anuais hipotéticas. Agora, perceba que por algum motivo, o ano em que as vendas ocorreram, só foram guardadas na primeira linha de cada ID (`id`). Isso é algo que devemos corrigir nessa tabela.

```
library(tidyverse)

v <- 2001:2004

set.seed(1)
df <- tibble(
  id = rep(1:4, each = 3),
  ano = NA_real_,
  valor = rnorm(12, mean = 1000, sd = 560)
)

df[seq(1, 12, by = 3), "ano"] <- v

df

## # A tibble: 12 x 3
##       id   ano valor
##   <int> <dbl> <dbl>
## 1     1  2001  649.
## 2     1    NA 1103.
## 3     1    NA  532.
```

```
## 4      2  2002 1893.
## 5      2    NA 1185.
## 6      2    NA  541.
## 7      3  2003 1273.
## 8      3    NA 1413.
## 9      3    NA 1322.
## 10     4  2004  829.
## 11     4    NA 1847.
## 12     4    NA 1218.
```

Portanto, o que queremos fazer, é completar as linhas de NA's, com o ano correspondente a essas vendas. Pelo fato dos anos estarem separados por um número constante de linhas, ou seja, a cada 3 linhas de NA's, temos um novo ano, podemos pensar em algumas soluções relativamente simples como a definida abaixo. Porém, a simplicidade do problema, depende dos intervalos entre cada valor, serem constantes. A partir do momento em que esses valores começarem a se dispersar em distâncias inconcistentes, uma solução como a definida abaixo, não servirá.

```
niveis <- unique(df$ano)
niveis <- niveis[!is.na(niveis)]
repair_vec <- df$ano
repair_vec[is.na(repair_vec)] <- rep(niveis, each = 2)

df$ano <- repair_vec
```

```
df
```

```
## # A tibble: 12 x 3
##       id   ano valor
##   <int> <dbl> <dbl>
## 1     1     1  2001  649.
## 2     1     1  2001 1103.
## 3     1     1  2001  532.
## 4     2     2  2002 1893.
## 5     2     2  2002 1185.
## 6     2     2  2002  541.
## 7     3     3  2003 1273.
## 8     3     3  2003 1413.
## 9     3     3  2003 1322.
## 10    4     4  2004  829.
## 11    4     4  2004 1847.
## 12    4     4  2004 1218.
```

Apesar de ser um problema simples, podemos alcançar uma solução ainda mais simples, ao utilizarmos funções que são especializadas nesses problemas. Esse é o caso da função `fill()` do pacote `tidyr`, que foi criada justamente para esse propósito. Portanto, sempre que você possuir em sua

tabela, uma coluna onde você deseja substituir uma sequência de NA's pelo último (ou próximo) valor disponível, você pode utilizar essa função para tal tarefa.

A função `fill()` possui três argumentos: 1) `data`, o objeto onde a tabela com que deseja trabalhar, está salva; 2) `...`, a lista de colunas em que você deseja aplicar a função; 3) `.direction`, define a direção que a função deve seguir na hora de preencher os valores.

```
library(tidyverse)
```

```
df %>% fill(ano)
```

```
## # A tibble: 12 x 3
##       id   ano valor
##   <int> <dbl> <dbl>
## 1     1    2001  649.
## 2     1    2001 1103.
## 3     1    2001  532.
## 4     2    2002 1893.
## 5     2    2002 1185.
## 6     2    2002  541.
## 7     3    2003 1273.
## 8     3    2003 1413.
## 9     3    2003 1322.
## 10    4    2004  829.
## 11    4    2004 1847.
## 12    4    2004 1218.
```

A função `fill()` trabalha a partir de uma dada direção vertical em sua tabela. Por padrão, a função `fill()` irá preencher os valores indo para baixo, ou seja, partindo do topo da tabela, até a sua base. Logo, a função irá substituir qualquer NA com o último valor disponível, ou em outras palavras, com o valor disponível anterior ao NA em questão. A função lhe oferece o argumento `.direction`, caso você deseja alterar esse comportamento. Logo, se você deseja preencher esses valores NA's com o próximo valor disponível em relação ao NA em questão. Isto é, preencher os valores para cima, partindo da base da tabela, e seguindo para o seu topo. Você precisa definir o argumento da seguinte maneira:

```
df %>% fill(ano, .direction = "up")
```

```
## # A tibble: 12 x 3
##       id   ano valor
##   <int> <dbl> <dbl>
## 1     1    2001  649.
## 2     1    2002 1103.
## 3     1    2002  532.
## 4     2    2002 1893.
```

```
## 5      2  2003 1185.
## 6      2  2003  541.
## 7      3  2003 1273.
## 8      3  2004 1413.
## 9      3  2004 1322.
## 10     4  2004  829.
## 11     4    NA 1847.
## 12     4    NA 1218.
```

```
# Caso prefira não utilizar o pipe ( %>% ),
# ficaria dessa forma:
```

```
# fill(df, ano, .direction = "up")
```

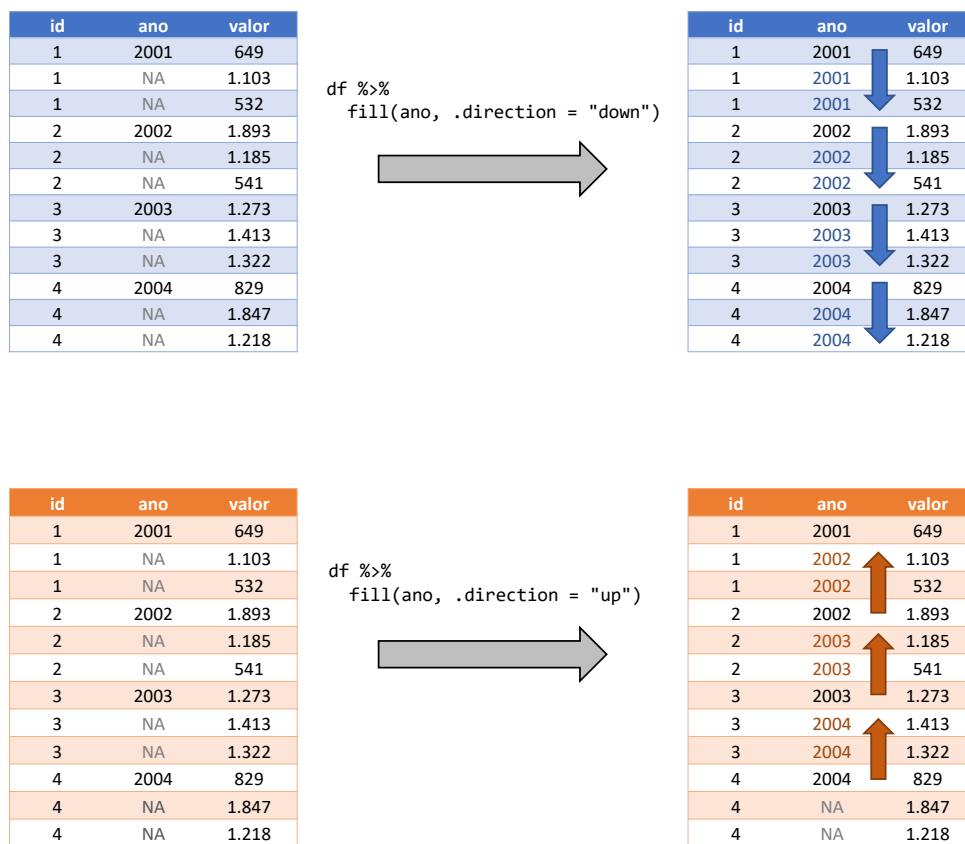
Portanto, se tivéssemos que colocar essas operações em uma representação visual, teríamos algo como a figura 7.5. Lembrando que a função usa por padrão, a direção *down*, logo, no primeiro caso mostrado na figura, você não precisaria definir explicitamente o argumento `.direction`.

Apesar de serem os exemplos mais claros de aplicação, serão raras as ocasiões em que você terá esse problema posto claramente já de início em sua tabela. Com isso, eu quero dizer que serão raros os momentos em que você desde o início terá uma tabela, onde por algum motivo, os registros aparecem apenas na primeira (ou na última) linha que diz respeito aquele registro.

Usualmente, você irá utilizar a função `fill()` quando você já estiver realizando diversas outras transformações em sua tabela, para se chegar aonde deseja. Um exemplo claro dessa ideia, seria uma tabela onde os valores são registrados no primeiro dia de cada semana (basicamente você possui dados semanais), mas você precisa calcular uma média móvel diária. Isso significa que para calcular essa média móvel, você teria que completar os dias faltantes de cada semana, e ainda utilizar o `fill()` para transportar o valor do primeiro dia, para os dias restantes da semana.

Vale ressaltar, que você pode utilizar em `fill()`, todos os mecanismos de seleção que introduzimos em `select()`, para selecionar as colunas em que você deseja aplicar a função `fill()`. Isso também significa, que com `fill()` você pode preencher várias colunas ao mesmo tempo. Agora, para relembrarmos esses mecanismos, vamos criar uma tabela inicialmente vazia, que contém o total de vendas realizadas nos 6 primeiros meses de 2020, por cada funcionário de uma loja.

```
set.seed(2)
funcionarios <- tibble(
  mes = rep(1:6, times = 4),
  vendas = floor(rnorm(24, mean = 60, sd = 24)),
  nome = NA_character_,
  salario = NA_real_,
  mes_ent = NA_real_,
  ano_ent = NA_real_,
  unidade = NA_character_
)
```

Figura 7.5: Representação do processo executado pela função fill**Fonte:** Elaboração própria do autor.

```
funcionarios
```

```
## # A tibble: 24 x 7
##   mes vendas nome  salario mes_ent ano_ent unidade
##   <int>  <dbl> <chr>    <dbl>   <dbl>   <dbl> <chr>
## 1     1     38 <NA>      NA      NA      NA <NA>
## 2     2     64 <NA>      NA      NA      NA <NA>
## 3     3     98 <NA>      NA      NA      NA <NA>
## 4     4     32 <NA>      NA      NA      NA <NA>
## 5     5     58 <NA>      NA      NA      NA <NA>
## 6     6     63 <NA>      NA      NA      NA <NA>
## 7     1     76 <NA>      NA      NA      NA <NA>
## 8     2     54 <NA>      NA      NA      NA <NA>
## 9     3    107 <NA>      NA      NA      NA <NA>
## 10    4     56 <NA>      NA      NA      NA <NA>
## # ... with 14 more rows
```

Em seguida, vamos preencher as colunas vazias (nome, salario, mes_ent, ...) de forma com que as informações de cada vendedor, apareçam apenas na última linha que diz respeito aquele vendedor. Como exemplo, as informações do vendedor Henrique, aparecem apenas na sexta linha da tabela, que é a última linha da tabela que se refere a ele.

```
valores <- list(
  salario = c(1560, 2120, 1745, 1890),
  nome = c("Henrique", "Ana", "João", "Milena"),
  ano_ent = c(2000, 2001, 2010, 2015),
  mes_ent = c(2, 10, 5, 8),
  unidade = c("Afonso Pena", "Savassi", "São Paulo", "Amazonas")
)
```

```
colunas <- colnames(funcionarios)[3:7]
```

```
for(i in colunas){
  funcionarios[1:4 * 6, i] <- valores[[i]]
}
```

```
# Com isso, temos o seguinte resultado:
funcionarios %>% print(n = 12)
```

```
## # A tibble: 24 x 7
##   mes vendas nome  salario mes_ent ano_ent unidade
```

```
##      <int> <dbl> <chr>      <dbl> <dbl> <dbl> <chr>
## 1      1      38 <NA>         NA     NA     NA <NA>
## 2      2      64 <NA>         NA     NA     NA <NA>
## 3      3      98 <NA>         NA     NA     NA <NA>
## 4      4      32 <NA>         NA     NA     NA <NA>
## 5      5      58 <NA>         NA     NA     NA <NA>
## 6      6      63 Henrique 1560      2    2000 Afonso Pena
## 7      1      76 <NA>         NA     NA     NA <NA>
## 8      2      54 <NA>         NA     NA     NA <NA>
## 9      3     107 <NA>         NA     NA     NA <NA>
## 10     4      56 <NA>         NA     NA     NA <NA>
## 11     5      70 <NA>         NA     NA     NA <NA>
## 12     6      83 Ana      2120     10    2001 Savassi
## # ... with 12 more rows
```

Portanto, o que precisamos é aplicar a função `fill()` usando `.direction = "up"`, em cada uma dessas colunas vazias, de forma a preencher o restante das linhas com as informações de cada vendedor. Dada a natureza dessa tabela, os dois melhores mecanismos que aprendemos em `select()`, para selecionarmos essas colunas vazias, são: 1) usar os índices dessas colunas; 2) nos basearmos nos tipos de dados contidos em cada coluna; 3) usar um vetor externo com os nomes das colunas que desejamos.

Para utilizar o método 3 que citei acima, podemos utilizar o vetor `colunas` que criamos agora a pouco ao preenchermos a tabela, e já contém os nomes das colunas que desejamos. Porém, para o exemplo abaixo do método 2, você talvez se pergunte: “Se estamos aplicando a função `fill()` sobre todas as colunas que contém ou dados de texto (`character`), ou dados numéricos (`numeric`), nós também estamos aplicando a função sobre as colunas `mes` e `vendas`, das quais não necessitam de ajuste. O que acontece?”. Nada irá ocorrer com as colunas `mes` e `vendas`, caso elas já estejam corretamente preenchidas, portanto, podemos aplicar a função sobre elas sem medo.

```
# Todas as três alternativas abaixo
# geram o mesmo resultado:
```

```
funcionarios %>%
  fill(
    3:7,
    .direction = "up"
  )
```

```
funcionarios %>%
  fill(
    all_of(colunas),
    .direction = "up"
  )
```

```
funcionarios %>%
```

```
fill(
  where(is.character),
  where(is.numeric),
  .direction = "up"
)
```

```
## # A tibble: 24 x 7
```

	mes	vendas	nome	salario	mes_ent	ano_ent	unidade
	<int>	<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<chr>
## 1	1	38	Henrique	1560	2	2000	Afonso Pena
## 2	2	64	Henrique	1560	2	2000	Afonso Pena
## 3	3	98	Henrique	1560	2	2000	Afonso Pena
## 4	4	32	Henrique	1560	2	2000	Afonso Pena
## 5	5	58	Henrique	1560	2	2000	Afonso Pena
## 6	6	63	Henrique	1560	2	2000	Afonso Pena
## 7	1	76	Ana	2120	10	2001	Savassi
## 8	2	54	Ana	2120	10	2001	Savassi
## 9	3	107	Ana	2120	10	2001	Savassi
## 10	4	56	Ana	2120	10	2001	Savassi

```
## # ... with 14 more rows
```

7.6 Um estudo de caso sobre médias móveis com complete() e fill()

7.6.1 A metodologia de uma média móvel no R

Uma média móvel é calculada ao aplicarmos o cálculo da média aritmética, sobre uma sequência de partes (ou *subsets*) de seus dados. De certa forma, esse processo se parece com uma rolagem, como se estivéssemos “rolando” o cálculo da média ao longo dos nossos dados. Veja por exemplo, o cálculo abaixo, onde utilizamos a função `roll_mean()` do pacote `RcppRoll` para calcularmos uma média móvel que possui uma janela de 3 valores.

```
library(RcppRoll)

vec <- c(2.7, 3.0, 1.5, 3.2, 1.6, 2.5)

roll_mean(vec, n = 3)

## [1] 2.400000 2.566667 2.100000 2.433333
```

A janela (ou *window*) de uma média móvel, representa o número de observações que serão utilizadas no cálculo da média a cada “transição”, ou a cada “rolagem”. No exemplo acima, aplicamos uma média móvel com uma janela de 3 valores. Isso significa que a cada “rolagem”, são utilizados 3

valores no cálculo da média. Na primeira rolagem, temos a média do vetor (2.7, 3.0, 1.5). Já na segunda rolagem, temos a média do vetor (3.0, 2.5, 3.2). E assim por diante. Portanto, em uma representação visual, o cálculo da média móvel aplicada por `roll_mean()`, é apresentado na figura 7.6.

Perceba também que o cálculo de uma média móvel implica em perda de observações. Pois no exemplo anterior, o vetor `vec` possui 6 valores, já o resultado de `roll_mean()` possui apenas 4 valores. Diversas operações estatísticas como essa, possuem o mesmo efeito. Um outro exemplo, seriam as operações de diferenciação, que são muito utilizadas em análises de séries temporais, e produzem essa mesma perda de informação. Por outro lado, no caso exposto aqui, essa perda de observações, ocorre devido ao tamanho da janela para o cálculo da média móvel.

Ou seja, pelo fato de que definimos uma janela de 3 observações para o cálculo da média móvel acima, as duas primeiras observações do vetor `vec`, não podem gerar a sua própria média móvel. Dito de outra forma, a função `roll_mean()` não pode calcular uma média nas duas primeiras rolagens sobre o vetor `vec`. Pois na primeira rolagem sobre o vetor, `roll_mean()` possui apenas uma observação (2.7). Já na segunda rolagem, `roll_mean()` acumula ainda duas observações (2.7, 3.0). Apenas a partir da terceira rolagem, que `roll_mean()` poderá calcular uma média segundo o tamanho da janela que definimos para ela, pois ela agora possui três observações (2.7, 3.0, 1.5) disponíveis para o cálculo. A partir daí, `roll_mean()` vai continuar rolando e calculando as médias móveis, até atingir o conjunto final de três observações do vetor `vec` (3.2, 1.6, 2.5).

Logo, sendo j o número de observações presentes em cada janela de cálculo de sua média móvel, e $lvec$ sendo o número de valores presentes em seu vetor inicial, sobre o qual você irá calcular a sua média móvel. O número de médias móveis resultantes de `roll_mean()` (obs), será equivalente a: $obs = lvec - (j - 1)$. Em outras palavras, o número de observações que você irá perder ($perda$), no cálculo de sua média móvel será equivalente a: $perda = j - 1$.

Agora, pode ser de seu desejo, contornar essa perda de observações de alguma maneira. Especialmente se você está calculando essa média móvel com base em uma coluna de sua tabela, pois sendo este o caso, provavelmente será de seu interesse, guardar essas médias calculadas em uma nova coluna dessa tabela. Entretanto, se você criar uma tabela, alocando por exemplo o vetor `vec` em uma coluna, e tentasse adicionar uma nova coluna contendo as médias móveis de cada ponto do vetor, o R lhe retornaria o erro abaixo. Caso você se lembre das propriedades dos `data.frame`'s no R, você irá entender o porquê que está motivando esse erro. Pois todas as colunas de um `data.frame` devem possuir obrigatoriamente o mesmo número de observações. Como nós perdemos duas das seis observações de `vec`, no cálculo da média móvel, o R não permite alocarmos diretamente essas médias em nossa tabela `df`.

```
df <- data.frame(x = vec)
```

```
df$media_movel <- roll_mean(df$x, n = 3)
```

```
Error in `df$<-data.frame`(`*tmp*`, media_movel, value = c(2.4,
2.566666666666667, : replacement has 4 rows, data has 6.
```

Figura 7.6: Representação do cálculo de uma média móvel

Vetor inicial	Função aplicada	Vetor resultante
2.7 3.0 1.5 3.2 1.6 2.5	→ <code>mean(c(2.7, 3.0, 1.5))</code> =	2.40
2.7 3.0 1.5 3.2 1.6 2.5	→ <code>mean(c(3.0, 1.5, 3.2))</code> =	2.566667
2.7 3.0 1.5 3.2 1.6 2.5	→ <code>mean(c(1.5, 3.2, 1.6))</code> =	2.10
2.7 3.0 1.5 3.2 1.6 2.5	→ <code>mean(c(3.2, 1.6, 2.5))</code> =	2.433333

Fonte: Elaboração própria do autor.

A função `roll_mean()` oferece duas formas de contornar esse problema: 1) definir o alinhamento da função, para preencher as observações faltantes com valores não-disponíveis (NA); 2) ou preencher essas observações faltantes com um valor pré-definido, através de seu argumento `fill`. Ambas formas são válidas e possivelmente são o que você deseja.

O primeiro método que citei, envolve o alinhamento da função, que você irá definir através dos sufixos `r` e `l` no nome da função. A diferença entre os dois tipos de alinhamento, decide em que parte (no início, ou no final) do vetor resultante de `roll_mean()`, os valores não-disponíveis (NA) serão posicionados. Se você deseja utilizar o alinhamento à direita (*right* - `r`), você deve utilizar a função `roll_meanr()`. Mas se você quer utilizar o alinhamento à esquerda (*left* - `l`), você deve utilizar a função `roll_meanl()`.

Quanto ao segundo método que citei, que envolve o argumento `fill`, você pode utilizar o argumento `align`, para definir em que partes do vetor resultante, o valor que você definiu em `fill` será posicionado. Por exemplo, caso eu use o valor *center* em `align`, o valor definido em `fill` vai aparecer tanto no início quanto ao fim do vetor que resulta da função `roll_mean()`. Mas se eu utilizar o valor *right* em `align`, esse valor irá aparecer ao início do vetor resultante.

```
roll_meanr(vec, n = 3)

## [1]      NA      NA 2.400000 2.566667 2.100000 2.433333

roll_meanl(vec, n = 3)

## [1] 2.400000 2.566667 2.100000 2.433333      NA      NA

roll_mean(vec, n = 3, fill = 0, align = "center")

## [1] 0.000000 2.400000 2.566667 2.100000 2.433333 0.000000

roll_mean(vec, n = 3, fill = 0, align = "right")

## [1] 0.000000 0.000000 2.400000 2.566667 2.100000 2.433333

roll_mean(vec, n = 3, fill = 0, align = "left")

## [1] 2.400000 2.566667 2.100000 2.433333 0.000000 0.000000
```

7.6.2 Os dados da Covid-19

Na próxima seção, busco dar um exemplo prático de como as funções `complete()` e `fill()` que vimos nas seções anteriores, podem ser utilizadas em conjunto em um problema real. Para isso, vamos utilizar parte dos dados sobre a Covid-19 (SARS-COV-2) no Brasil. Ao utilizar o código abaixo, lembre-se de renomear a primeira coluna da tabela para `dia`. Dessa forma, nós evitamos confusões com qualquer função que possua um argumento chamado `data` (funções como `mutate()`, `select()`, `complete()`, `lm()` e muitas outras possuem tal argumento).

```
library(tidyverse)

github <- "https://raw.githubusercontent.com/pedropark99/"
arquivo <- "Curso-R/master/Dados/covid.csv"

covid <- read_csv2(paste0(github, arquivo))

colnames(covid)[1] <- "dia"
```

A Fundação João Pinheiro (FJP-MG) tem dado apoio técnico ao governo de Minas Gerais, no acompanhamento da pandemia de COVID-19, ao gerar estatísticas e estimações epidemiológicas para o estado. Eu fiz parte desse esforço por algum tempo, e uma demanda real que havia chegado para mim na época, consistia no cálculo de uma média móvel dos novos casos diários da doença para cada estado do Brasil. Pois era de desejo da Secretaria Estadual de Saúde, comparar a curva dessa média móvel do estado de Minas Gerais, com a de outros estados brasileiros.

Na época em que trabalhei com a base covid, ela possuía algumas barreiras, que superei com o uso de `complete()` e `fill()`. São essas barreiras, e suas resoluções que busco mostrar nessa seção, como um exemplo real de uso dessas funções. Porém, a base covid que está disponível hoje, e que você acaba de importar através dos comandos acima, é a base já corrigida e reformatada e, por isso, ela já se encontra em um formato ideal para o cálculo de uma média móvel. Portanto, antes de partirmos para a prática, vou aplicar algumas transformações, com o objetivo de “corromper” a base covid até o seu ponto inicial. Pois o foco nessa seção, se encontra na demonstração dos problemas de formatação da base, e em suas possíveis soluções.

O primeiro ponto a ser discutido, são as datas iniciais da pandemia em cada estado brasileiro. No Brasil, a pandemia de Covid-19 atingiu primeiramente o estado de São Paulo, e chegou posteriormente aos demais estados. Como podemos ver pelo resultado abaixo, a data inicial de cada estado, ao longo da base covid é difusa. Em alguns estados, os registros se iniciam a partir da data do primeiro registro de casos da doença (como os estados do Acre, Alagoas, Bahia, Amazonas e Espírito Santo). Alguns estados, registraram mais de um caso já no primeiro dia (como o Acre, que registrou três casos no dia 17 de Março, e o Ceará, que reportou nove casos no dia 16 de Março). Porém, outros estados (como a Paraíba) não seguem esse padrão, pois no seu primeiro dia de registro, o número de casos reportados foi igual a zero. Ou seja, a pandemia no estado da Paraíba não se iniciou oficialmente no dia 12 de Março, pois não havia casos reportados até este dia.

```
data_inicial <- covid %>%
  group_by(estado) %>%
  summarise(
    data_inicial = min(dia),
    casos_inicial = min(casos)
  )

data_inicial %>% print(n = 15)
```

```
## # A tibble: 27 x 3
##   estado data_inicial casos_inicial
##   <chr>   <date>           <dbl>
## 1 AC     2020-03-17             3
## 2 AL     2020-03-08             1
## 3 AM     2020-03-13             2
## 4 AP     2020-03-20             1
## 5 BA     2020-03-06             1
## 6 CE     2020-03-16             9
## 7 DF     2020-03-07             1
## 8 ES     2020-03-05             1
## 9 GO     2020-03-12             3
## 10 MA    2020-03-20             1
## 11 MG     2020-03-08             1
## 12 MS     2020-03-14             2
## 13 MT     2020-03-20             1
## 14 PA     2020-03-18             1
## 15 PB     2020-03-12             0
## # ... with 12 more rows
```

Isso não representa um grande problema, mas antes das próximas transformações, devemos iniciar os dados de cada estado, no dia de primeiro registro de casos da doença. Isto é, os dados da Paraíba, por exemplo, devem se iniciar no dia em que houve pela primeira vez, um registro de casos maior do que zero. Como a coluna casos, representa o número acumulado de casos da doença, nós podemos realizar esse “nivelamento” entre os estados, ao eliminarmos da base, todas as linhas que possuem um número acumulado de casos igual a zero. Porém, vale a pena olharmos mais atentamente sobre essas linhas antes de eliminá-las, para termos certeza de que não estamos causando mais danos ao processo.

Perceba pelo resultado abaixo, que todas as linhas em que o número acumulado de casos se iguala a zero, pertencem ao estado da Paraíba. Todas essas seis datas, são “inúteis” para o propósito da base covid, pois apresentam um cenário anterior à pandemia no estado da Paraíba. Portanto, antes de prosseguirmos, vamos eliminar essas linhas, com o uso de `filter()`.

```
covid %>% filter(casos == 0)
```

```
## # A tibble: 6 x 4
##   dia          estado casos mortes
##   <date>      <chr>   <dbl> <dbl>
## 1 2020-03-12 PB           0      0
## 2 2020-03-13 PB           0      0
## 3 2020-03-14 PB           0      0
## 4 2020-03-15 PB           0      0
## 5 2020-03-16 PB           0      0
## 6 2020-03-17 PB           0      0
```

```
covid <- filter(covid, casos != 0)
```

A base covid atualmente possui os números de casos diários acumulados de cada estado brasileiro. Mas vamos supor, que a base covid registrasse o número de casos acumulados, somente nos dias em que esse número se alterasse. Ou seja, se o número de casos acumulados da doença em uma segunda-feira qualquer do ano, era de 300, e esse número se manteve constante ao longo da semana, até que na sexta-feira, esse número subiu para 301 casos, a base covid irá registrar os números de casos acumulados apenas para as datas da segunda e da sexta dessa semana. Tal resultado pode ser atingido com os comandos abaixo. Como nós filtramos anteriormente a base, de forma a retirar as linhas com valores iguais a zero na coluna casos, temos que reconstruir o objeto data_inicial, como exposto abaixo.

```
data_inicial <- covid %>%
  group_by(estado) %>%
  summarise(
    data_inicial = min(dia),
    casos_inicial = min(casos)
  ) %>%
  mutate(
    dia = as.Date(data_inicial - 1),
    casos = NA_real_,
    mortes = NA_real_
  )

covid <- covid %>%
  bind_rows(
    data_inicial %>% select(dia, estado, casos, mortes)
  ) %>%
  group_by(estado) %>%
  arrange(
    dia,
    estado,
    .by_group = TRUE
  ) %>%
  mutate(
    teste = lead(casos) == casos
  ) %>%
  filter(teste == FALSE) %>%
  ungroup() %>%
  select(-teste)
```

Portanto, temos agora a tabela abaixo, onde podemos perceber que no dia 21 de Março de 2020 não houve alteração no número de casos acumulados no estado do Acre. Pois esse dia (2020-03-21) não está mais presente na tabela covid. Dito de outra forma, o número de novos casos de Covid-19

que surgiram no dia 21 de Março, foi igual a zero. O mesmo ocorre com os dias 25 e 27 de Março no estado, que também não estão mais presentes na base.

covid

```
## # A tibble: 3,456 x 4
##   dia          estado casos mortes
##   <date>      <chr>  <dbl> <dbl>
## 1 2020-03-18 AC           3      0
## 2 2020-03-19 AC           4      0
## 3 2020-03-20 AC           7      0
## 4 2020-03-22 AC          11      0
## 5 2020-03-23 AC          17      0
## 6 2020-03-24 AC          21      0
## 7 2020-03-26 AC          23      0
## 8 2020-03-28 AC          25      0
## 9 2020-03-29 AC          34      0
## 10 2020-03-30 AC          41      0
## # ... with 3,446 more rows
```

Quando trabalhei anteriormente com a base covid anteriormente, ela se encontrava inicialmente em um formato muito próximo deste. Por isso, a base necessitava de ajustes para o cálculo da média móvel. O intuito da próxima seção, é demonstrar como eu fiz esses ajustes necessários, através das funções `complete()` e `fill()`.

7.6.3 Buscando soluções com `complete()` e `fill()`

Considerando que você aplicou as transformações expostas na seção anterior (**Os dados da Covid-19**), você está apto a aplicar os comandos apresentados nessa seção. Agora, em que sentido essa nova tabela que temos, é inapropriada para o cálculo da média móvel diária de novos casos? Porque agora faltam os registros dos dias em que não houve alteração no número acumulado de casos em cada estado. Ou seja, nós retiramos na seção anterior, justamente aquilo que queremos recuperar nessa seção. Como eu disse, o intuito dessas seções, está nos exemplos de uso das funções `complete()` e `fill()`, e não no caminho que temos que percorrer para estarmos aptos para a aplicação desses exemplos.

Portanto, o problema que possuímos agora no cálculo da média móvel sobre a base covid, é que faltam os dias onde o número de casos acumulados permaneceu constante. Isso significa que agora temos uma quebra no cálculo da média móvel. Caso o vetor `vec` abaixo, representasse uma parte da coluna `casos` da nossa base covid, se aplicássemos uma média móvel, com uma janela de 3 valores, as médias dos dias 03 e 04 não poderiam ser calculadas (ou no mínimo, estariam incorretas), tendo em vista as transformações que aplicamos na seção anterior.

Ou seja, considerando que nós eliminamos na seção anterior, todas as linhas de covid, onde o número acumulado de casos permaneceu constante em relação ao seu valor anterior; se aplicarmos

a mesma transformação ao vetor `vec` abaixo, o valor referente ao dia 02 seria eliminado, e por isso, uma quebra ocorreria sobre o cálculo das médias móveis dos dias 03 e 04. Pois, dentre os valores dos 3 dias anteriores aos dias 03 e 04, estaria faltando o valor referente ao dia 02.

```
vec <- c("Dia 01" = 1, "Dia 02" = 1, "Dia 03" = 3,
        "Dia 04" = 5, "Dia 05" = 7)
```

```
vec
```

```
## Dia 01 Dia 02 Dia 03 Dia 04 Dia 05
##      1      1      3      5      7
```

São por essas razões, que devemos recuperar os dias perdidos na tabela `covid`, mesmo que o número de casos nesses dias tenham permanecido constantes. No nosso caso, não podemos utilizar diretamente as colunas da base `covid`, para expandirmos a tabela, e recuperarmos as datas que foram perdidas. Pois essas datas não se encontram mais na tabela `covid`. Lembre-se que a função `complete()` irá sempre trabalhar com as observações que estão presentes em sua base, caso você não forneça algo a mais, com a qual ela possa trabalhar. Por isso, teremos que gerar na função `complete()`, um vetor externo à base `covid`, de forma a incluirmos todas as datas que faltam.

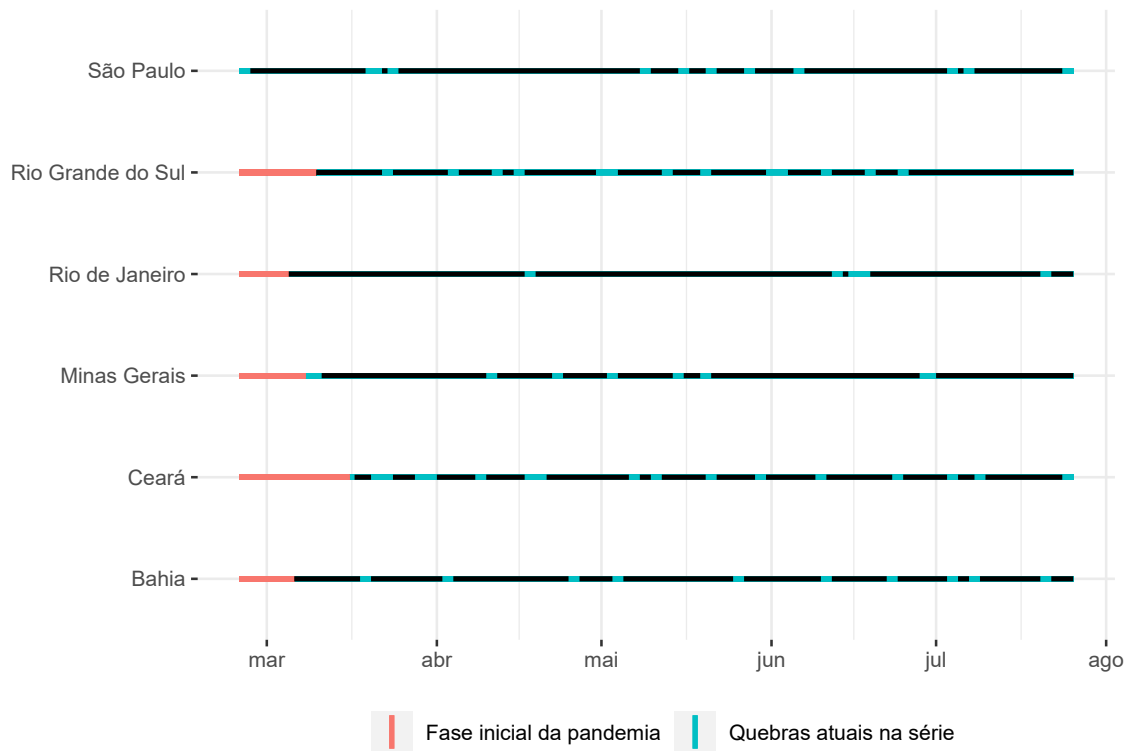
Antes de prosseguir, vamos compreender exatamente qual é o estado atual da base `covid`. Nós eliminamos (na seção anterior) parte dos dias da base. Mais especificamente aqueles dias em que o número acumulado de casos da doença, permaneceu constante em relação a seu valor anterior. Portanto, neste momento, as séries temporais do número de casos de cada estado apresentam quebras. Em uma representação visual, essas séries se assemelham no momento às linhas em cor preta, apresentadas no gráfico da figura 7.7. São essas quebras que nos impedem de calcularmos uma média móvel desses casos.

O primeiro passo, será expandir essas séries com a função `complete()`. Utilizando-se de um vetor (construído pela função `seq.Date()`), contendo desde o dia 1 da pandemia no país (dia dos primeiros casos no estado de São Paulo, onde a pandemia se iniciou) até o último dia da base. Com isso, a função `complete()` irá combinar cada uma dessas datas, com cada um dos 27 estados. Após essa expansão da tabela `covid`, as séries de cada estado vão incluir todas as datas possíveis, incluindo aquelas que originalmente não pertenciam aquele estado. Dessa forma, as séries de cada estado, vão ser equivalentes à junção das linhas pretas, azuis e vermelhas no gráfico da figura 7.7. Formando assim novamente uma série “sólida”, ou completa.

```
menor_data <- min(data_inicial$data_inicial)
maior_data <- max(covid$dia)

novo_covid <- covid %>%
  complete(
    dia = seq.Date(menor_data, maior_data, by = "day"),
    estado
  ) %>%
```

Figura 7.7: Representação das séries temporais da base covid pré e pós-transformações



Fonte: Elaboração própria do autor.

```
group_by(estado) %>%
  arrange(dia, estado, .by_group = T) %>%
  ungroup()
```

```
novo_covid
```

```
## # A tibble: 4,131 x 4
##   dia      estado casos mortes
##   <date>    <chr>  <dbl> <dbl>
## 1 2020-02-25 AC      NA     NA
## 2 2020-02-26 AC      NA     NA
## 3 2020-02-27 AC      NA     NA
## 4 2020-02-28 AC      NA     NA
## 5 2020-02-29 AC      NA     NA
## 6 2020-03-01 AC      NA     NA
## 7 2020-03-02 AC      NA     NA
## 8 2020-03-03 AC      NA     NA
## 9 2020-03-04 AC      NA     NA
## 10 2020-03-05 AC      NA     NA
## # ... with 4,121 more rows
```

O segundo passo, será “nivelar” as séries de acordo com o período inicial de cada estado. Pois, como resultado do passo anterior, as séries de todos os estados serão iguais em comprimento (ou em número de observações). Pois as séries de todos os estados, estarão incluindo desde o dia 1 da pandemia, até o último dia da pandemia. Portanto, seguindo o gráfico da figura 7.7, no segundo passo estaremos eliminando a área vermelha de cada série, de forma que as séries de cada estado vão se equivaler à junção das linhas em preto e azul. Para isso, podemos aplicar os comandos abaixo:

```
novo_covid <- novo_covid %>%
  right_join(
    data_inicial[c("estado", "data_inicial")],
    by = "estado"
  )

teste <- novo_covid$dia >= novo_covid$data_inicial

novo_covid <- novo_covid[teste, ]

novo_covid

## # A tibble: 3,670 x 5
##   dia      estado casos mortes data_inicial
##   <date>    <chr>  <dbl> <dbl> <date>
## 1 2020-03-17 AC      NA     NA 2020-03-17
## 2 2020-03-18 AC      3      0 2020-03-17
```



```
## 3 2020-03-19 AC      4      0 2020-03-17
## 4 2020-03-20 AC      7      0 2020-03-17
## 5 2020-03-21 AC     NA     NA 2020-03-17
## 6 2020-03-22 AC     11      0 2020-03-17
## 7 2020-03-23 AC     17      0 2020-03-17
## 8 2020-03-24 AC     21      0 2020-03-17
## 9 2020-03-25 AC     NA     NA 2020-03-17
## 10 2020-03-26 AC     23      0 2020-03-17
## # ... with 3,660 more rows
```

O terceiro passo, envolve o uso de `fill()` para completarmos o número de casos em cada data recuperada. Lembre-se que ao expandirmos a tabela com `complete()`, a função preencheu os campos das colunas casos e mortes com valores não-disponíveis (NA), na linha de cada data que não estava presente anteriormente na base covid (ou seja, as datas que foram perdidas anteriormente). Portanto, todas as linhas que possuem um valor NA nessas colunas, são as linhas que correspondem aos dias em que o número acumulado de casos se manteve constante. Como esse número se manteve constante, tudo o que precisamos fazer, é utilizar `fill()` para puxar os valores disponíveis anteriores para esses campos.

```
novo_covid <- novo_covid %>%
  fill(casos, mortes, .direction = "up")

novo_covid

## # A tibble: 3,670 x 5
##   dia          estado casos mortes data_inicial
##   <date>      <chr>  <dbl> <dbl> <date>
## 1 2020-03-17 AC        3      0 2020-03-17
## 2 2020-03-18 AC        3      0 2020-03-17
## 3 2020-03-19 AC        4      0 2020-03-17
## 4 2020-03-20 AC        7      0 2020-03-17
## 5 2020-03-21 AC       11      0 2020-03-17
## 6 2020-03-22 AC       11      0 2020-03-17
## 7 2020-03-23 AC       17      0 2020-03-17
## 8 2020-03-24 AC       21      0 2020-03-17
## 9 2020-03-25 AC       23      0 2020-03-17
## 10 2020-03-26 AC       23      0 2020-03-17
## # ... with 3,660 more rows
```

Dessa forma, temos novamente, a tabela corretamente formatada, e pronta para o cálculo de uma média móvel. O número acumulado de casos certamente tende a aumentar com o tempo, mas será que a variação desse número, segue o mesmo padrão? Para calcularmos essa variação, podemos utilizar a função `lag()` para utilizarmos o valor da linha anterior de uma coluna. Com isso, podemos subtrair o valor da linha anterior, sobre o valor da linha atual, tirando assim, a diferença ou a

variação entre elas. Em seguida, basta aplicarmos a função `roll_meanr()` sobre esta variação, para adquirirmos uma média móvel do número de novos casos diários.

```
library(RcppRoll)

novo_covid <- novo_covid %>%
  group_by(estado) %>%
  mutate(
    novos_casos = casos - lag(casos),
    media_casos = roll_meanr(novos_casos, n = 5)
  )

novo_covid

## # A tibble: 3,670 x 7
## # Groups:   estado [27]
##   dia      estado casos mortes data_inicial novos_casos media_casos
##   <date>    <chr>  <dbl>  <dbl> <date>          <dbl>      <dbl>
## 1 2020-03-17 AC         3      0 2020-03-17         NA         NA
## 2 2020-03-18 AC         3      0 2020-03-17         0         NA
## 3 2020-03-19 AC         4      0 2020-03-17         1         NA
## 4 2020-03-20 AC         7      0 2020-03-17         3         NA
## 5 2020-03-21 AC        11      0 2020-03-17         4         NA
## 6 2020-03-22 AC        11      0 2020-03-17         0          1.6
## 7 2020-03-23 AC        17      0 2020-03-17         6          2.8
## 8 2020-03-24 AC        21      0 2020-03-17         4          3.4
## 9 2020-03-25 AC        23      0 2020-03-17         2          3.2
## 10 2020-03-26 AC        23      0 2020-03-17         0          2.4
## # ... with 3,660 more rows
```

```
t <- "Média móvel de 5 dias para os novos casos de Covid-19 nos estados
da região Sudeste"
```

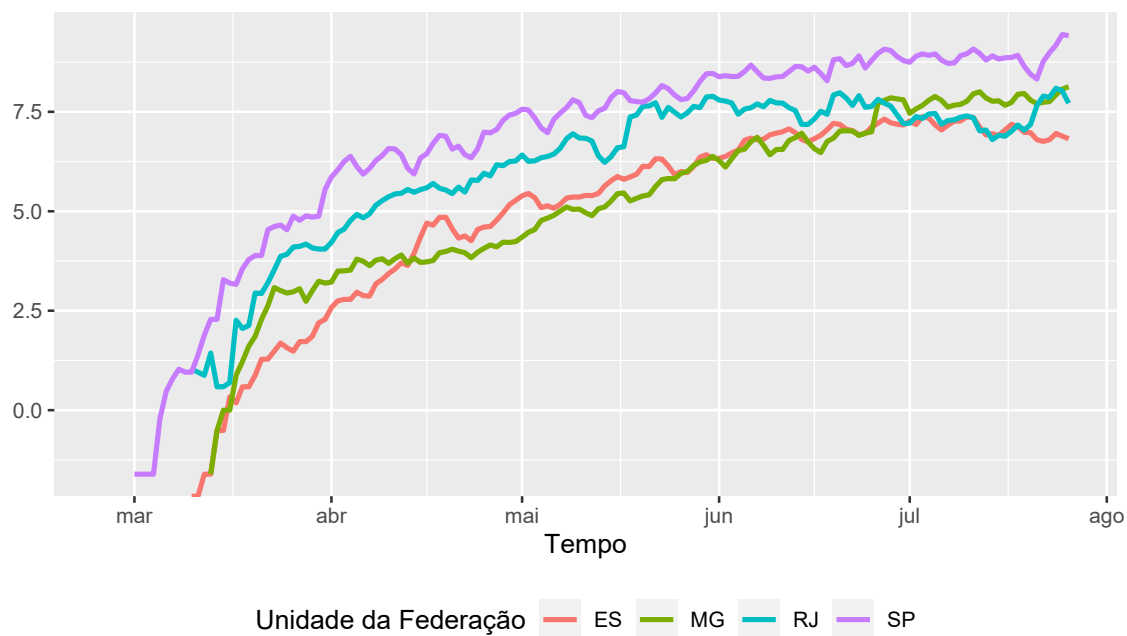
```
novo_covid %>%
  filter(estado %in% c("SP", "MG", "RJ", "ES")) %>%
  ggplot() +
  geom_line(
    aes(x = dia, y = log(media_casos), color = estado),
    size = 1
  ) +
  theme(
    legend.position = "bottom",
    axis.title.y = element_blank(),
    plot.title = element_text(face = "bold")
  ) +
```

```
labs(  
  title = t,  
  subtitle = "Escala logarítmica",  
  x = "Tempo",  
  color = "Unidade da Federação"  
)
```

```
## Warning: Removed 20 row(s) containing missing values (geom_path).
```

Média móvel de 5 dias para os novos casos de Covid-19 nos estados da região Sudeste

Escala logarítmica



Fonte: Elaboração própria do autor.

Exercícios

Exercício 1

Os itens desta questão vão utilizar a tabela `world_bank_pop`. Essa tabela advém do pacote `tidyr`, logo, caso você tenha chamado com sucesso por esse pacote através do comando `library()` você já possui acesso a essa tabela. A tabela `world_bank_pop` contém uma série histórica de vários dados populacionais para cada país descrito na base.

```
world_bank_pop

## # A tibble: 1,056 x 20
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2006`
##   <chr>    <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 ABW     SP.URB.TO~  4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4
## 2 ABW     SP.URB.GR~  1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2
## 3 ABW     SP.POP.TO~  9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5
## 4 ABW     SP.POP.GR~  2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1
## 5 AFG     SP.URB.TO~  4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6
## 6 AFG     SP.URB.GR~  3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0
## 7 AFG     SP.POP.TO~  2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7
## 8 AFG     SP.POP.GR~  3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0
## 9 AGO     SP.URB.TO~  8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7
## 10 AGO    SP.URB.GR~  5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0
## # ... with 1,046 more rows, and 11 more variables: 2007 <dbl>, 2008 <dbl>,
## #   2009 <dbl>, 2010 <dbl>, 2011 <dbl>, 2012 <dbl>, 2013 <dbl>,
## #   2014 <dbl>, 2015 <dbl>, 2016 <dbl>, 2017 <dbl>
```

- A) A tabela `world_bank_pop` não se encontra em um formato *tidy*. Indique qual (ou quais) dos pressupostos que definem o formato *tidy data* é (ou são) violado por essa tabela e, explique o porquê disso.
- B) Repare que para além das colunas `country` e `indicator`, temos os dados populacionais espalhados ao longo de diversas colunas, onde cada coluna representa o valor dessa série histórica para um determinado ano. Utilize os conhecimentos desse capítulo para reunir essas várias colunas (que se referem a anos específicos da série) de modo que a base fique mais próxima de um formato *tidy data*.
- C) Filtre todas as linhas da tabela que descrevem a população total de cada país (isto é, as linhas em que o valor na coluna `indicator` é igual ao código "SP.POP.TOTL"), em seguida, tente calcular a variação da população total entre cada ano da série, para todos os países.

Capítulo 8

Visualização de dados com ggplot2

8.1 Introdução e pré-requisitos

Esse é o primeiro capítulo em que vamos introduzir o pacote `ggplot2`, ou simplesmente `ggplot`. O `ggplot` é um pacote voltado para a visualização de dados, ou em outras palavras, para a construção de gráficos. Para que você possa acompanhar os exemplos dessa seção, você precisa ter o pacote instalado em sua máquina. Após instalá-lo, você pode tanto chamar diretamente pelo pacote `ggplot2`, quanto pelo `tidyverse`, que também o inclui, através da função `library()`.

```
library(ggplot2)
library(tidyverse)
```

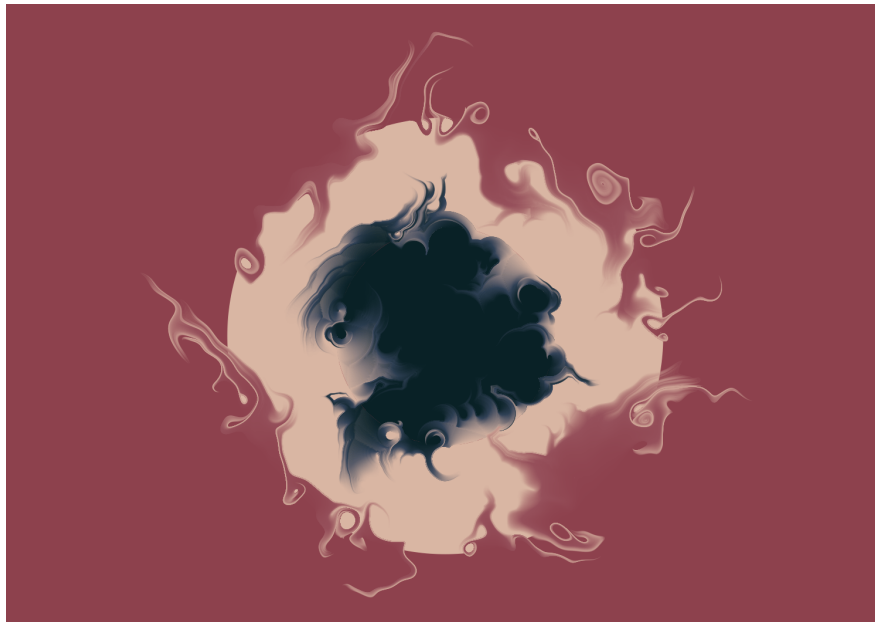
8.2 O que é o ggplot e a sua gramática

A linguagem R é conhecida por sua capacidade gráfica, e [Murrell \(2006\)](#) oferece ótimos exemplos que atestam essa afirmação. Mesmo que a linguagem R ofereça “já de fábrica”, o pacote `lattice`, que já é capaz de muita coisa, o `ggplot` é sem dúvidas, o pacote mais popular da linguagem no que tange a criação de gráficos, pois ele oferece algo que os outros pacotes não tem, que é a sua *flexibilidade*¹.

Flexibilidade é uma das principais características (e a principal vantagem) do `ggplot`, e é o que amplia a sua utilização para além dos gráficos sérios e frios de um jornal científico, permitindo ao usuário criar gráficos vistosos, e um tanto peculiares. Veja por exemplo, a arte criada por Thomas Lin Pedersen, mostrada na figura 8.1. O que lhe dá essa liberdade dentro do `ggplot`, é a sua gramática.

O pacote `ggplot` (ou seu nome oficial - `ggplot2`) foi inicialmente desenvolvido por [Wickham \(2016\)](#), e lançado pela primeira vez no ano de 2005. O pacote representa uma implementação da

¹Ou seja, em relação a seus concorrentes, o `ggplot` não é necessariamente o pacote que oferece praticidade, mas sim, um leque de possibilidades muito maior em relação a seus concorrentes.

Figura 8.1: phases9032 por Thomas Lin Pedersen

Fonte: Generative Art by Thomas Lin Pedersen².

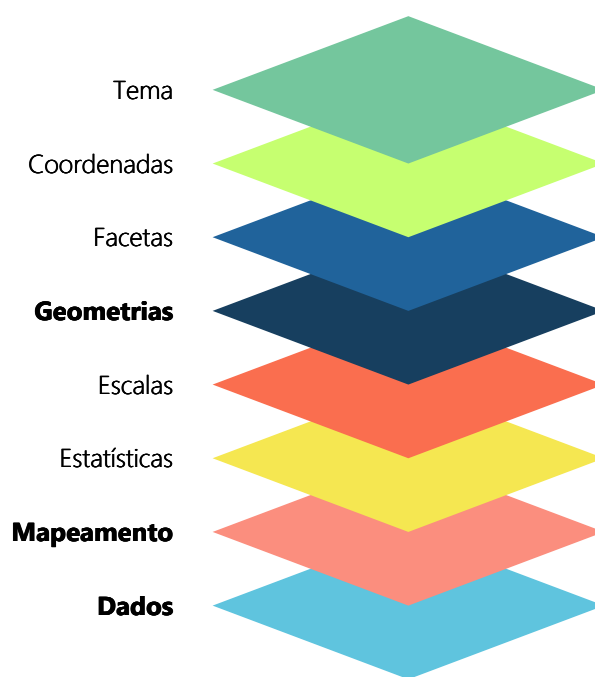
teoria desenvolvida por [Wilkinson \(2005\)](#), chamada de *The Grammar of Graphics* (ou “a gramática dos gráficos”). Portanto, o ggplot busca abstrair os principais conceitos da teoria de [Wilkinson \(2005\)](#), e implementá-los dentro da linguagem R.

Segundo a teoria de [Wilkinson \(2005\)](#), todo e qualquer gráfico estatístico, pode ser descrito por um conjunto de camadas, ou componentes, que estão apresentados na figura 8.2. Dessa forma, segundo a visão de [Wilkinson \(2005\)](#) todos os tipos de gráfico que conhecemos (pizza, barras, dispersão, *boxplot*, etc.) fazem parte de um mesmo grupo, e a característica que os tornam diferentes entre si, se encontra na forma como as camadas abaixo estão definidas em cada gráfico.

Tendo isso em mente, um gráfico do ggplot é composto por várias camadas, que em conjunto formam o gráfico desejado. A ideia por trás do pacote, portanto, é utilizar uma gramática para descrever de forma concisa, cada uma das camadas apresentadas na figura 8.2. Após definirmos essas camadas, nós podemos somá-las para construirmos o nosso gráfico final. Em outras palavras, você vai adicionando aos poucos, novas camadas ao gráfico, onde cada uma dessas camadas fica responsável por definir um componente específico do gráfico (escalas, formas geométricas, legendas, facetas, anotações, ...). Caso seja de seu desejo, você pode deixar o próprio ggplot responsável por definir várias das camadas apresentadas na figura 8.2. Porém, em todo gráfico do ggplot, você deve obrigatoriamente definir as três camadas (em negrito na figura 8.2) apresentadas a seguir, sendo portanto, as camadas **essenciais** que formam a base de todo gráfico do ggplot.

1. Dados: os dados que o gráfico deve expor.

Figura 8.2: Camadas de um gráfico do ggplot, baseado em Wickham (2016)



Fonte: Elaboração própria do autor. [WICKHAM, 2016](#), p. 4.

2. Mapeamento estético (*aesthetic mapping*): uma descrição de como as variáveis dispostas em seus dados devem ser mapeadas para elementos visuais (ou estéticos) de seu gráfico.
3. Geometrias: são as formas geométricas do gráfico que representam os seus dados, ou seja, em um gráfico de dispersão, seus dados são representados no gráfico por *pontos*, enquanto em um gráfico de barras, seus dados são representados por *retângulos*.

A gramática do ggplot, representa portanto, as regras que definem o emprego das funções necessárias, e de seus possíveis parâmetros para acessarmos e controlarmos cada uma das camadas mostradas na figura 8.2. Logo, cada uma dessas camadas, são controladas por uma função (ou por um conjunto de funções) diferente, que lhe permite o uso de diferentes mecanismos e valores em sua definição.

8.3 Iniciando um gráfico do ggplot

8.3.1 Dados

Primeiro, vamos definir os dados que vamos utilizar em nossos gráficos. A tabela `datasus`, contém a contagem de mortes por homicídios dolosos em 2018 no Brasil, coletados a partir dos microdados do SIM/DATASUS. Nessa mesma tabela, temos a distribuição dessas mortes, por sexo, por faixa etária, pelo estado (Unidade da Federação - UF) em que as mortes ocorreram, e pela cor de pele do indivíduos.

```
library(readr)
```

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

```
datasus <- read_csv2(paste0(github, pasta, arquivo))
```

```
## i Using '\',\'' as decimal and '\'.\' as grouping mark. Use `read_delim()` for more control.
```

```
##
## -- Column specification -----
## cols(
##   `Faixa etaria` = col_character(),
##   Genero = col_character(),
##   Cor = col_character(),
##   `Nome UF` = col_character(),
##   UF = col_character(),
##   Contagem = col_double()
## )
```

```
datasus
```



```
## # A tibble: 1,836 x 6
##   `Faixa etaria` Genero   Cor      `Nome UF` UF      Contagem
##   <chr>          <chr>   <chr>   <chr>    <chr>    <dbl>
## 1 10 a 14        Feminino Parda   Acre     AC        4
## 2 10 a 14        Masculino Parda   Acre     AC        4
## 3 15 a 19        Feminino Branca  Acre     AC        2
## 4 15 a 19        Feminino Parda   Acre     AC        4
## 5 15 a 19        Masculino Branca  Acre     AC        6
## 6 15 a 19        Masculino Parda   Acre     AC       65
## 7 15 a 19        Masculino Preta   Acre     AC        1
## 8 20 a 24        Feminino Indígena Acre     AC        1
## 9 20 a 24        Feminino Parda   Acre     AC        4
## 10 20 a 24       Masculino Branca  Acre     AC        7
## # ... with 1,826 more rows
```

Vamos começar a montar o nosso gráfico. Você **sempre** inicia um gráfico de ggplot, pela função base do pacote - `ggplot()`. Essa função é responsável por criar o objeto base para o gráfico, e nela, possuímos dois argumentos que compõe duas das três camadas essenciais (que definimos na seção 8.2) desse gráfico, e que podem ou não ser fornecidos nessa função. Esses dois argumentos são: *data*, que é o nome da tabela onde estão os dados que serão utilizados no gráfico; e *mapping*, que é o *aesthetic mapping*, ou o mapeamento de variáveis de sua tabela, para componentes estéticos do gráfico. Ou seja, você não precisa necessariamente fornecer esses argumentos já na função `ggplot()`, pois você pode definí-los dentro das funções que formam as figuras geométricas (as funções *geom*). O importante, é que você sempre deve começar um gráfico ggplot, com a função `ggplot()`.

Mas então, qual é a diferença entre eu fornecer esses argumentos na função `ggplot()` ou dentro das funções *geom*? Pense em um exemplo, no qual você busca mostrar em um mesmo gráfico, duas informações diferentes. Você pode utilizar dois *geom*'s (ou formas geométricas) diferentes para mostrar e diferenciar essas duas informações no gráfico. Por exemplo, podemos ter um gráfico que contenha barras mostrando a evolução da dívida pública, e linhas mostrando a evolução do salário médio no país.

Caso você forneça os dois argumentos (*data* e *mapping*) na função `ggplot()`, você está dizendo ao programa, que ele deve utilizar a mesma base de dados, e o mesmo *aesthetic mapping*, em todos os formatos geométricos (*geom*) do gráfico. Enquanto, ao fornecer esses argumentos dentro de cada função *geom*, você estaria dizendo ao programa que utilize essa base de dados, e esse *aesthetic mapping*, apenas naquele formato geométrico (ou *geom*) especificamente. Tendo isso em mente, não conseguiríamos montar o gráfico descrito no parágrafo anterior, ao dar os argumentos já na função `ggplot()`. Pois o gráfico mostra duas informações diferentes (salário médio e dívida pública), ao longo dos *geom*'s do gráfico. Ou seja, os dois formatos geométricos dispostos no gráfico, utilizam dois *aesthetic mapping* diferentes. Quando chegarmos à seção 8.4 vou explicar isso em mais detalhes.

No nosso caso, os dados que vamos utilizar, estão na tabela `datasus`, por isso forneço ao argumento

data o nome dessa tabela. Ao rodar o código logo abaixo, você vai perceber que ele gera apenas um quadro cinza vazio. Isso ocorre porque definimos apenas uma das camadas essenciais para compor o gráfico, que são os dados utilizados. Temos que definir as outras duas, para completarmos a base de um gráfico, por isso, vamos passar agora para o *aesthetic mapping*.

```
ggplot(data = datasus)
```

8.3.2 Mapeamento de variáveis (*Aesthetic Mapping*)

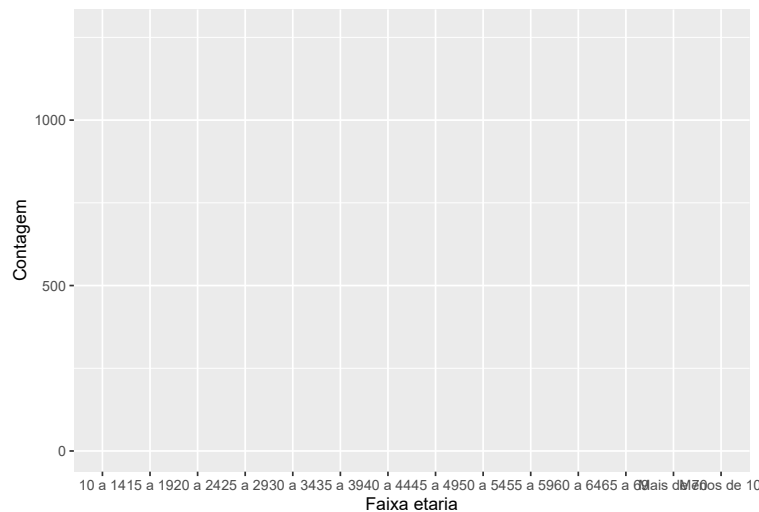
O *aesthetic mapping* representa o mapeamento, ou a conexão de variáveis em sua tabela (ou sua base de dados), com os componentes estéticos e visuais do gráfico. Nós controlamos esse mapeamento através da função `aes()`. Há diversos desses componentes visuais que compõe um gráfico, e os primeiros que vêm a sua mente, são provavelmente as cores e as formas geométricas. Mas também os próprios eixos, ou melhor dizendo, as escalas utilizadas nos eixos, são componentes visuais do gráfico. Pois essas escalas definem como as formas geométricas vão se posicionar dentro do espaço do gráfico.

Pense por exemplo, no globo terrestre. Você pode representar esse globo dentro do ggplot, mas para que os formatos sejam corretamente posicionados em um “globo”, você precisa usar uma escala e um sistema de coordenadas diferentes do plano cartesiano. Um outro exemplo, seria o gráfico de pizza. Ao pesquisar sobre, você irá perceber que um gráfico de pizza no ggplot, é feito a partir do formato geométrico de barras (`geom_bar()`). Ou seja, um gráfico de barras, é o ponto de partida para gerar um gráfico de pizza no ggplot, e o que diferencia esses dois gráficos, é a escala usada. Em um gráfico de pizza, utilizamos uma coordenada circular, chamada de coordenada polar, ao invés do plano cartesiano.

Agora, vamos continuar montando nosso gráfico. Primeiro, vamos tentar montar um gráfico de barras, que mostre a distribuição do total de mortes ao longo das faixas etárias no país, baseado nos dados apresentados na tabela `datasus`. Tendo isso em mente, o número de mortes, deve ficar no eixo y de nosso gráfico, enquanto os grupos (faixas etárias), devem ficar no eixo x.

Essa é a base do nosso mapeamento de variáveis. Estamos definindo que o número de mortes deve ficar no eixo y, e as faixas etárias no eixo x, e nós concedemos essa descrição ao ggplot, dentro da função `aes()` (abreviação para *aesthetic mapping*). Dessa vez, ao rodar o código abaixo você vai perceber que um plano cartesiano foi montado, onde temos uma escala para a faixa etária no eixo x, e outra escala para o total de mortes no eixo y. Porém, esse plano cartesiano continua vazio, pois ainda não definimos a última camada essencial do gráfico, que é a forma geométrica que deve representar os nossos dados.

```
ggplot(  
  data = datasus,  
  mapping = aes(x = 'Faixa etaria', y = Contagem)  
)
```



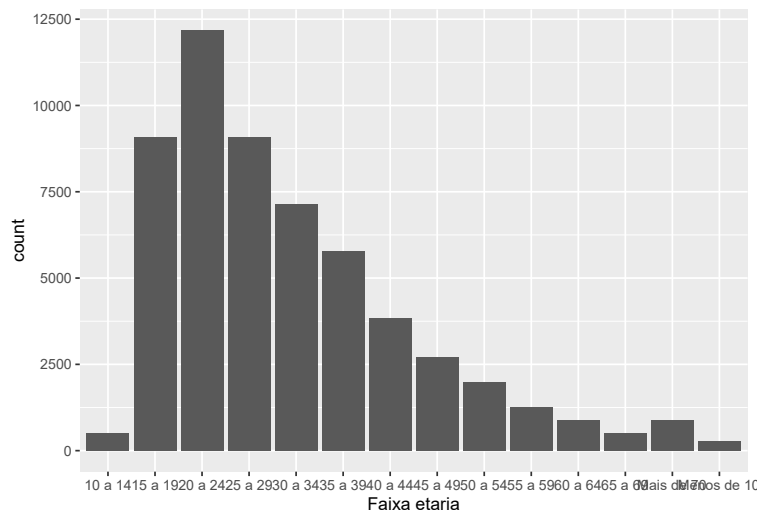
Fonte: Elaboração própria do autor.

Portanto, vamos adicionar ao nosso código, nossa primeira função geom. Cada função geom, se refere a um formato geométrico diferente. No nosso caso, queremos um gráfico de barras, que é formado pela função `geom_bar()`. O padrão dessa função (ou formato geométrico) no ggplot, é calcular uma contagem dos dados. Em outras palavras, o gráfico de barras no ggplot, se comporta inicialmente (por padrão) como um histograma. Ao invés de calcular a soma total de certa variável, ele irá contar, quantas vezes cada valor ocorre naquela variável dentro da base de dados.

Entretanto, não queremos uma contagem dos dados, pois a coluna `Contagem` já representa uma contagem em si. O que queremos é a soma total dessa contagem em cada faixa etária. Por isso, ao invés de fornecer `Contagem` ao argumento `y`, eu defino essa coluna para o argumento `weight`. Todas as funções que adicionarmos às várias camadas do nosso gráfico, devem ser conectadas por um sinal de `+`, por isso lembre-se de colocar esse sinal sempre que adicionar uma nova função ao seu gráfico.

```
ggplot(
  data = datasus,
  mapping = aes(x = 'Faixa etaria', weight = Contagem)
) +
  geom_bar()
```

Agora que definimos as três camadas essenciais (dados, *aesthetic mapping* e geom), temos enfim o nosso primeiro gráfico montado. Há várias coisas que poderíamos fazer a partir daqui. Podemos por exemplo, colorir as barras de acordo com a participação do sexo no número de mortes. Ou seja, essas cores irão representar em cada barra, o número de mortes que correspondem ao sexo masculino e ao sexo feminino. Por padrão, o `geom_bar()` empilha esses agrupamentos um em cima do outro. Dessa forma, essas cores não apenas nos apresenta o número de mortes em cada



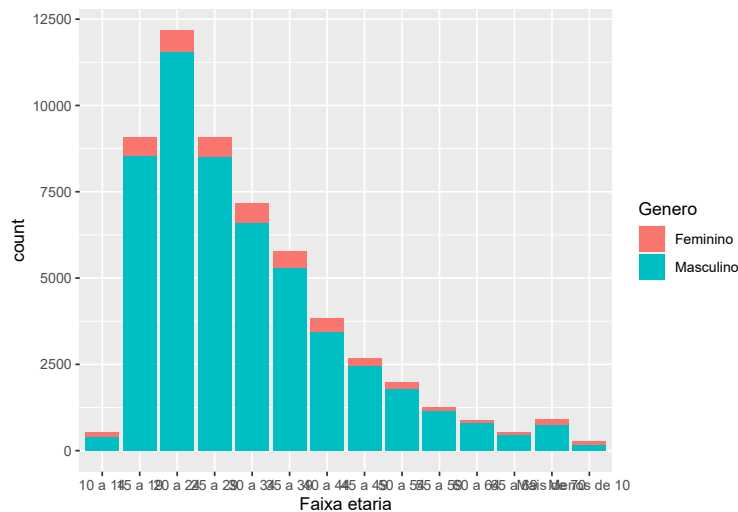
Fonte: Elaboração própria do autor.

sexo, mas indiretamente, elas também nos mostra o quanto que aquele grupo representa (qual a sua porcentagem) do total de mortes naquela faixa etária.

Desta maneira, estamos definindo um outro componente visual do gráfico (cores das barras) à uma outra variável de nossos dados (coluna `Genero`). Logo, estamos falando novamente do *aesthetic mapping* do gráfico, e por isso, devemos definir essa ligação dentro da função `aes()`. Há duas formas de você colorir formas geométricas no `ggplot`, que dependem da forma geométrica e do resultado que você quer atingir. Uma barra (ou retângulo), é tratada como uma forma geométrica de área, enquanto outras formas (como pontos e linhas) são tratadas de uma maneira diferente. Nesses formatos de área, você deve utilizar o argumento `fill`, para preencher o interior deste formato de uma cor.

```
ggplot(  
  data = datasus,  
  mapping = aes(  
    x = 'Faixa etaria',  
    weight = Contagem,  
    fill = Genero  
  )  
) +  
  geom_bar()
```

Conseguimos colorir as barras, e podemos ver que uma parte muito pequena das mortes correspondem ao sexo feminino, em todas as faixas etárias. Agora, e se mudássemos a variável de grupos (faixas etárias) do nosso gráfico? Vamos conectar uma variável diferente ao eixo x, por exemplo, a cor de pele. Perceba, que o restante do *aesthetic mapping* continua o mesmo, e portanto, o gráfico mantém essas outras “conexões” enquanto modifica a variável ligada ao eixo x.



Fonte: Elaboração própria do autor.

```
ggplot(
  data = datasus,
  mapping = aes(
    x = Cor,
    weight = Contagem,
    fill = Genero
  )
) +
geom_bar()
```

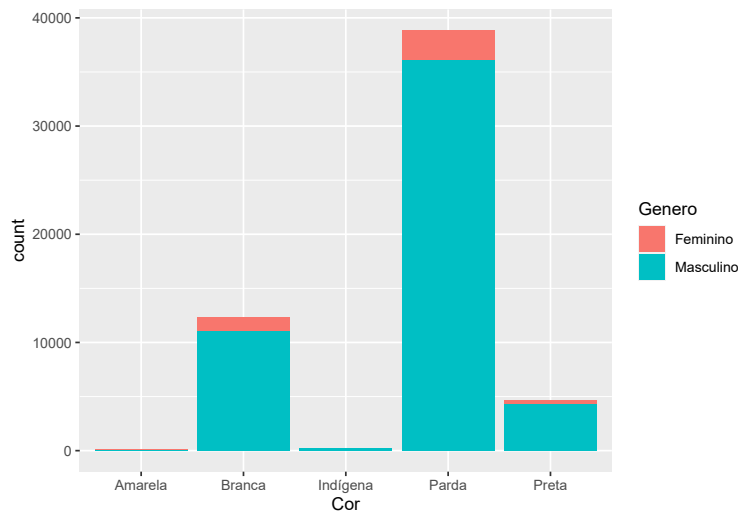
Como disse anteriormente, há outros componentes visuais que você pode ligar às variáveis de sua tabela de dados. Você pode por exemplo, em alguns geom's, conectar o *formato* desse geom a uma variável. Eu sei, sua cabeça provavelmente deu uma volta com esse exemplo: “Como assim? Eu posso variar o formato de uma forma geométrica ao longo do gráfico?”. Na seção 8.4 dou um exemplo dessa estratégia.

8.3.3 Formatos geométricos - funções geom

Cada função geom utiliza um formato geométrico e uma forma de desenho diferentes, para desenhar e representar os seus dados. No ggplot há vários geom's distintos que você pode utilizar. Abaixo estou listando os geom's mais comuns, mas basta consultar o painel de dicas do RStudio³, ou o site oficial de referências do pacote⁴, que você ficará um pouco perdido com tantas opções. Um excelente repositório, com ótimos exemplos de gráficos dos quais você pode tirar inspirações de

³<<https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>>

⁴<<https://ggplot2.tidyverse.org/reference/index.html>>



Fonte: Elaboração própria do autor.

como e onde utilizar cada geom, é o *R Graph Gallery*⁵.

1. `geom_bar()`: desenha um gráfico de barras.
2. `geom_point()`: desenha um gráfico de pontos (ou um gráfico de dispersão).
3. `geom_line()`: desenha um gráfico de linha.
4. `geom_boxplot()`: desenha um gráfico de *boxplot*.
5. `geom_histogram()`: desenha um histograma.
6. `geom_sf()`: desenha um mapa (geom para dados espaciais).
7. `geom_smooth()`: desenha uma linha de média condicional (muito utilizado para desenhar linhas que representam modelos de regressão linear e de outros modelos econométricos).
8. `geom_text()`: utilizado para inserir texto.
9. `geom_label()`: utilizado para inserir rótulos, ou basicamente, textos envoltos por uma caixa.

Por exemplo, um gráfico de barras, é geralmente utilizado para apresentar estatísticas descritivas dos nossos dados. Ou seja, esse tipo de gráfico (por padrão) tenta resumir características dos seus dados em poucos números (médias, totais, contagens). Já um gráfico de dispersão (por padrão) nos apresenta diretamente os dados, de forma crua no plano cartesiano. Isto é, diferentes geom's vão tratar (e principalmente, representar) os seus dados de formas distintas. Vamos por exemplo, adicionar pontos ao nosso gráfico anterior, com o `geom_point()`.

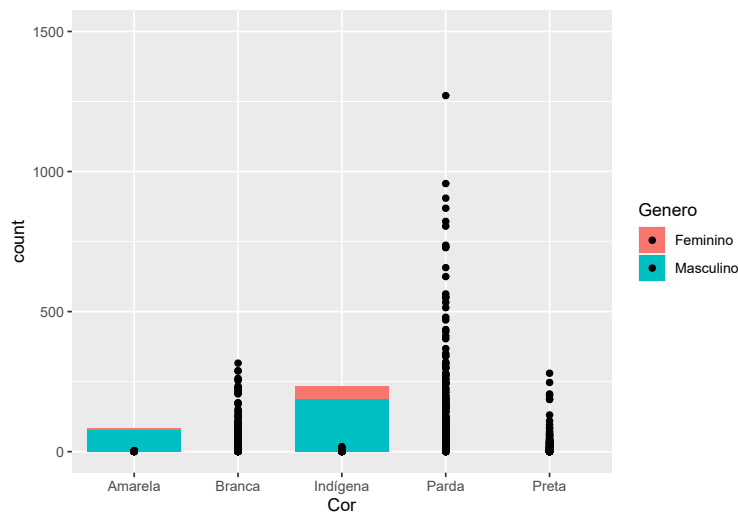
Para facilitar a visualização, eu limitei os valores do eixo y no gráfico (para o intervalo de 0 a 1500)

⁵<<https://www.r-graph-gallery.com/>>

por meio da função `lims()`. Dessa forma, estamos dando um *zoom* em uma área específica do gráfico. Repare que cada ponto representa uma das observações encontradas na nossa base, e que vários deles estão se sobrepondo.

```
ggplot(
  data = datasus,
  mapping = aes(
    x = Cor,
    weight = Contagem,
    fill = Genero
  )
) +
geom_bar() +
geom_point(aes(y = Contagem)) +
lims(y = c(0,1500))
```

Warning: Removed 6 rows containing missing values (geom_bar).



Fonte: Elaboração própria do autor.

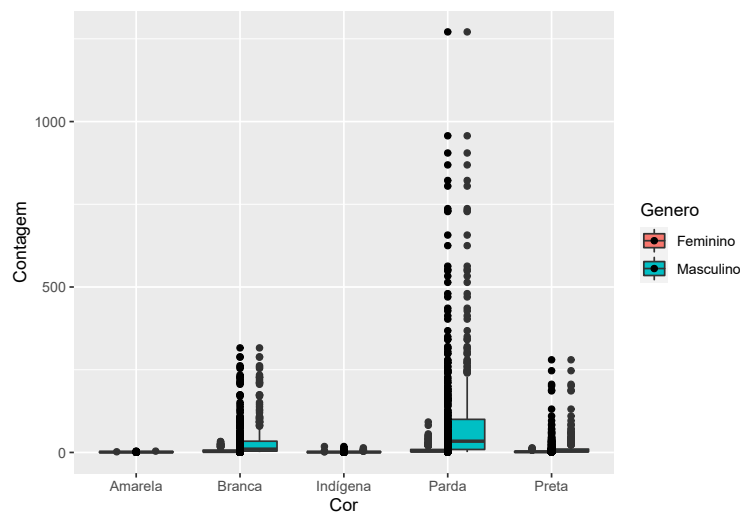
Ao substituírmos as barras por *boxplot*'s produzimos um gráfico que além de mostrar todas observações da base com o `geom_point()`, ele também mostra como a distribuição de ambos os sexos se encaixam ao longo do alcance (ou *range*) desses dados. Podemos perceber que nos indivíduos de cor parda, a maior contagem para o sexo feminino em toda a base, atinge em torno de 175 mortes, enquanto para o sexo masculino, esses valores podem atingir mais de 1000 mortes, apesar de que ambos os valores são *outliers* em suas respectivas distribuições.

```
ggplot(
  data = datasus,
```

```

mapping = aes(
  x = Cor,
  y = Contagem,
  fill = Genero
)
) +
geom_boxplot() +
geom_point()

```



Fonte: Elaboração própria do autor.

Uma outra forma de visualizarmos a diferença entre homens e mulheres nesses dados, seria utilizando geom's de erro, como as linhas de alcance. Os geom's de erro são muito úteis para visualizar medidas de variação ao longo dos grupos. O `geom_ribbon()` por exemplo, é utilizado em gráficos de séries temporais, para desenhar os intervalos de confiança ou desvios padrão ao redor da linha que representa a série. No nosso caso, iremos utilizar o `geom_linerange()`, para desenharmos a diferença média entre o número de mortes entre os dois gêneros.

O que o `geom_linerange()` faz é desenhar uma reta de um ponto A a um ponto B. A ideia por trás desse geom é desenharmos uma linha que representa (pelo seu comprimento), por exemplo, o desvio padrão de uma variável, ou no nosso caso, a diferença na média de vítimas de homicídios dolosos entre dois gêneros. Isto significa que temos dois novos componentes visuais que podemos controlar no gráfico através do *aesthetic mapping*, que são as coordenadas do ponto A e do ponto B. Esses componentes (pontos A e B) representam os “limites” (máximo e mínimo) dessa linha, por isso, são controlados pelos argumentos `ymin` e `ymax` dentro da função `aes()`. Há outros geom's que podem ser controlados por esses argumentos, porém os que vimos anteriormente (`geom_point()` e `geom_bar()`) não possuem esses argumentos.

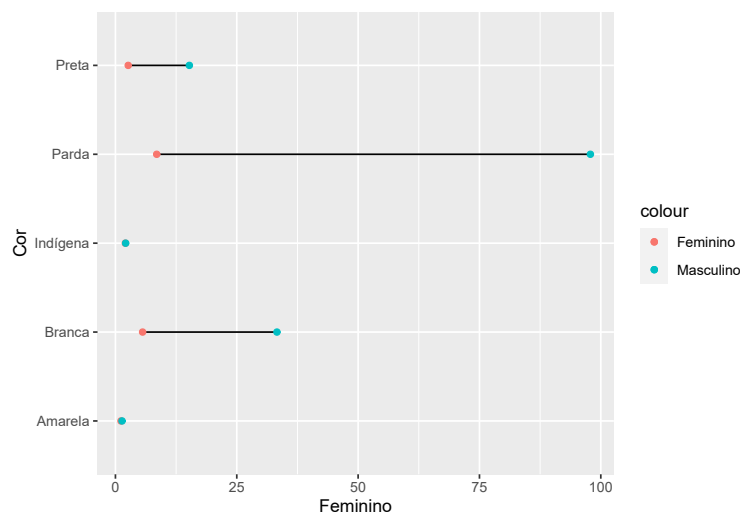
Primeiro, precisamos calcular o número médio de mortes de cada gênero e em cada cor de pele, e

em seguida, modificar a estrutura da tabela, para que possamos mapear os limites (ymin e ymax) do *linrange* de acordo com o sexo. Para isso utilizo funções do pacote *dplyr*, portanto lembre-se de chamar por esse pacote com *library()*, para ter acesso a essas funções. Perceba também que eu invertei o plano cartesiano, utilizando a função *coord_flip()*.

```
datasus_agrup <- datasus %>%
  group_by(Cor, Genero) %>%
  summarise(Media = mean(Contagem)) %>%
  pivot_wider(
    id_cols = c("Cor", "Genero"),
    names_from = "Genero",
    values_from = "Media"
  )
```

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

```
ggplot(
  data = datasus_agrup,
  aes(x = Cor)
) +
  geom_linerange(aes(ymax = Masculino, ymin = Feminino)) +
  geom_point(aes(y = Feminino, color = "Feminino")) +
  geom_point(aes(y = Masculino, color = "Masculino")) +
  coord_flip()
```



Fonte: Elaboração própria do autor.

Agora, muitas coisas estão ocorrendo neste gráfico. Primeiro, o *geom_linerange()* constrói uma linha para cada cor de pele, que vai da média de mortes no sexo feminino até a média no sexo

masculino. Segundo, dois `geom_point()` são utilizados, onde cada um deles fica responsável por um dos sexos, e desenha um único ponto para cada cor de pele que indica a média de mortes para o sexo correspondente. Em seguida, eu uso `coord_flip()` para inverter o plano cartesiano. Ou seja, a variável que estava no eixo y (média de mortes) vai para o eixo x, e a variável que estava no eixo x (cor de pele) vai para o eixo y.

Certamente, esse gráfico dá um pouco mais de trabalho de construir. Porém, é uma forma mais simples de se mostrar essa diferença, e com isso, você consegue atingir um público maior. Pode ser que o seu leitor não saiba o que é um *boxplot*, e há razões razoáveis para se acreditar nisso. No Brasil, o *boxplot* não é comumente tratado no ensino básico, e sim no ensino superior, e mais especificamente, em cursos que sejam da área de exatas, ou que possuam matérias de estatística na grade curricular. Nós sabemos também que o acesso da população brasileira ao ensino superior é restrito, sendo considerado um local de “elitismo”.

Por outro lado, os alunos em geral veem as principais medidas estatísticas de posição central (média, mediana e moda) já no ensino básico, e alguns chegam a revê-las no ensino superior. Logo, as chances de seu leitor compreender a mensagem que você quer passar: “em média, os homens são as principais vítimas de homicídios dolosos, entretanto, nas populações indígenas e de cor de pele amarela, esse padrão não parece ser significativo” são maiores. Essa consideração pode ter menor peso a depender de qual seja o público que você busca atingir. Se você está publicando um artigo científico em sua área, é bem provável que os potenciais leitores deste material conheçam um *boxplot*, e portanto, saibam interpretá-lo corretamente.

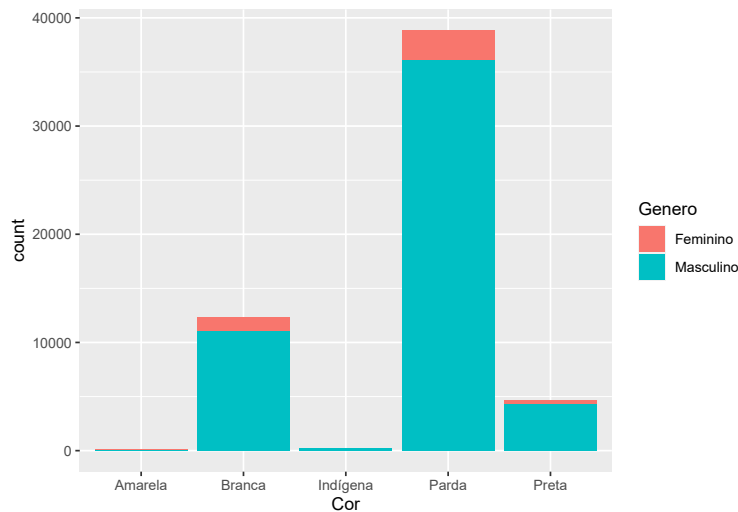
8.4 Uma outra forma de se compreender o *aesthetic mapping*

Nas seções anteriores, eu defini o *aesthetic mapping*, como a conexão entre as variáveis de sua tabela, com os componentes visuais de seu gráfico. Porém, temos uma outra forma útil de enxergarmos esse sistema. Podemos entender o *aesthetic mapping*, como um mecanismo para determinarmos quais componentes visuais vão variar, e quais vão permanecer constantes ao longo do gráfico. Ou seja, se você está definindo, por exemplo, as cores da forma geométrica (*geom*) que representa os seus dados, você pode utilizar o *aesthetic mapping* para definir se e como essas cores vão variar ao longo do gráfico.

Por exemplo, vamos voltar ao gráfico de barras que montamos na seção 8.3, que mostra o número total de mortes ao longo das diferentes cores de pele e gênero da base. Perceba, que a cor está variando dentro de cada barra (e não entre cada uma delas), de acordo com a variável *Genero*.

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor, weight = Contagem, fill = Genero)  
  )
```

Nós podemos modificar a forma como essas cores variam dentro de cada barra, ao mudarmos a



Fonte: Elaboração própria do autor.

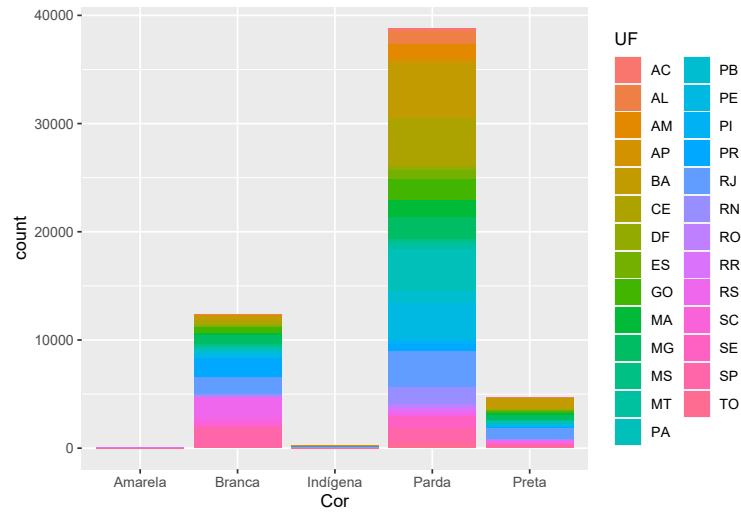
variável que define essa variação. Em outras palavras, podemos alterar o comportamento das cores, ao conectar esse componente em `aes()`, a uma outra variável de nossa tabela. Como exemplo, podemos atribuir às UF's da base. Perceba que temos agora, uma variação muito maior de cores dentro de cada barra.

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor, weight = Contagem, fill = UF)  
  )
```

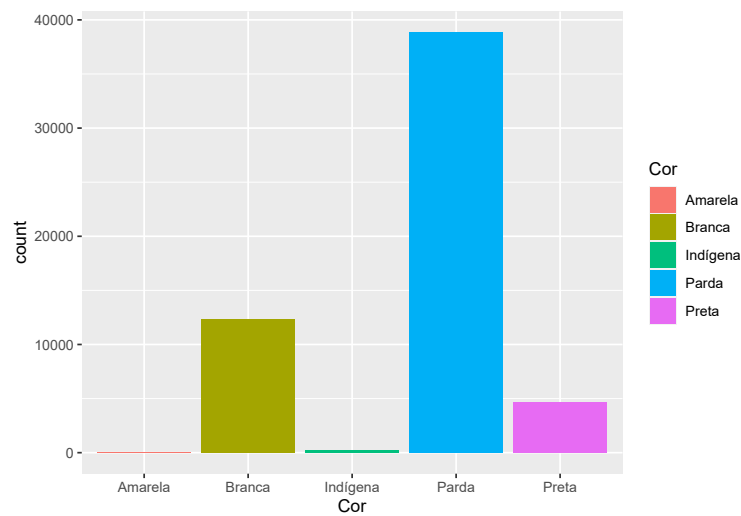
Nós podemos ainda, atribuir a mesma variável alocada no eixo x para definir a variação dessas cores ao longo do gráfico. Dessa forma, temos um gráfico onde cada uma das barras terá a sua própria cor. Isso não é particularmente útil, mas talvez você deseja ter uma cor separada para cada barra, e caso você esteja com preguiça de pensar e definir quais cores serão essas, deixar essa tarefa nas mãos do próprio ggplot é uma solução e um atalho simples para atingir um bom resultado.

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor, weight = Contagem, fill = Cor)  
  )
```

Portanto, ao conectarmos diferentes variáveis com o argumento (`fill`) em `aes()`, que define como as cores de cada barra são compostas, podemos modificar a forma como essas cores variam ao longo do gráfico. Mas e se nós quisermos manter uma única cor para essas barras, ou seja, e se é de seu



Fonte: Elaboração própria do autor.



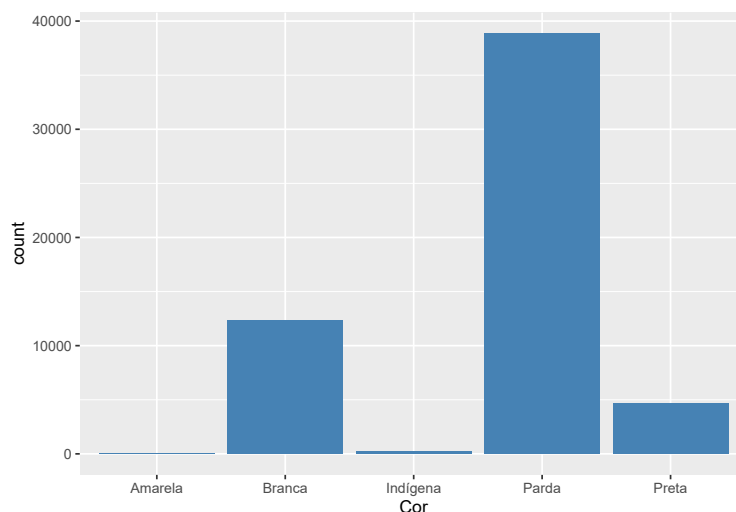
Fonte: Elaboração própria do autor.

desejo manter as cores constantes ao longo de todo o gráfico? Para isso, basta que você defina essas cores, fora de `aes()`.

Em outras palavras, a função `aes()` trabalha com **variáveis**, ou atributos de suas observações que tendem a variar ao longo de sua base. Quando você estiver trabalhando com valores constantes, ou com atributos que possuem um único valor possível ao longo de toda a sua base, a função `aes()` provavelmente não será o lugar ideal para trabalharmos com tais valores.

Por exemplo, o R possui diversas cores pré-programadas em seu sistema, e sempre que você quiser acessar essas cores ao longo do `ggplot`, você pode se referir a elas, através de seus nomes registrados. Caso queira uma lista com os nomes dessas cores pré-programadas, você pode utilizar a função `colors()`. Dentre essas diversas cores, temos uma chamada de `steelblue`. Logo, caso eu queira que todas as barras do meu gráfico estejam coloridas de acordo com essa cor, eu preciso fornecer o nome dessa cor ao argumento `fill`, de fora da função `aes()`.

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor, weight = Contagem),  
    fill = "steelblue"  
  )
```



Fonte: Elaboração própria do autor.

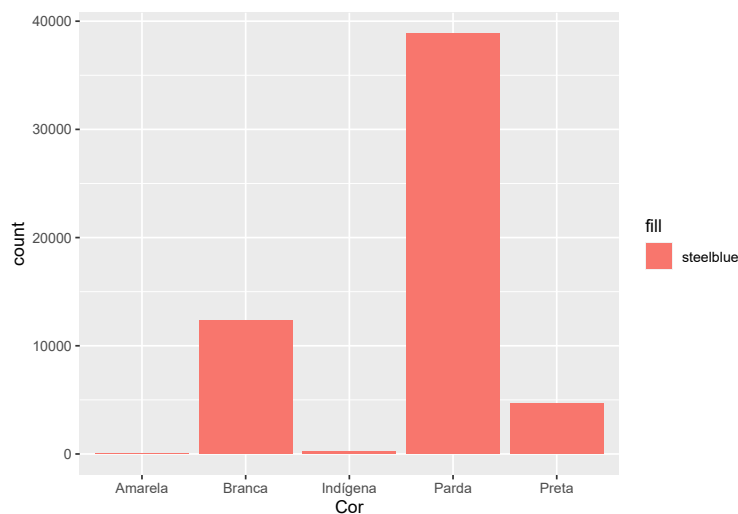
Portanto, você pode aplicar essa metodologia para qualquer outro componente visual de seu gráfico que pode ser definido. Ou seja, se você deseja manter algum dos componentes visuais, definidos por um dos argumentos de `aes()` (`fill`, `color`, `size`, `fontface`, `linetype`, `shape`, etc.), constantes, você precisa apenas definir esses argumentos fora de `aes()`. Por outro lado, caso você queira

controlar a forma como algum desses componentes visuais variam ao longo do gráfico, você precisa definir esses argumentos dentro de `aes()`.

Porém, você talvez se pergunte: o que ocorre se eu fornecer a cor `steelblue` dentro de `aes()`? Será que o `ggplot` reconhece que queremos aplicar essa cor sobre as formas geométricas do gráfico? A resposta curta é não, mas o resultado em geral é um pouco estranho, ou no mínimo algo inesperado. Pois em um caso como esse, a função `aes()` irá entender que você deseja colorir as barras no gráfico, de acordo com uma nova variável em sua tabela, chamada `fill`, e que possui um único valor possível ao longo da base, mais especificamente, o texto `steelblue`.

Esse comportamento ocorre sempre que você fornece um valor em texto (um *string*) à algum argumento de `aes()`. Em uma situação como essa, o `ggplot()` parece criar uma nova variável em sua tabela chamada `fill`, e que contém o valor em texto que você forneceu a esse argumento. Isso não necessariamente é um comportamento inadequado, mas ele certamente surpreende alguns usuários, e como ele tem se mantido ao longo das últimas versões do `ggplot`, é possível que ele continue a funcionar dessa forma, por um bom tempo.

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor, weight = Contagem, fill = "steelblue")  
  )
```



Fonte: Elaboração própria do autor.

8.5 Sobrepondo o *aesthetic mapping* inicial em diversas camadas

Agora, vou explicar em maiores detalhes qual é a diferença entre: preencher os argumentos de data e mapping já na função inicial do gráfico (`ggplot()`), e de preenchê-los nas funções `geom`.

Para isso, vamos usar outros dados. Na tabela PIB eu possuo uma série histórica mensal do índice de faturamento real da indústria (`Faturamento_indus`), da porcentagem do PIB que representa a dívida pública líquida (`Divida_liq_perc`), e a média mensal da taxa de investimento produtivo (taxa de formação bruta de capital fixo - FBCF) na economia brasileira, além de dados de PIB, coletados do IPEAD⁶.

PIB

```
## # A tibble: 184 x 6
##   Data          PIB PIB_acumulado Divida_liq_perc FBCF Faturamento_indus
##   <date>      <dbl>      <dbl>          <dbl> <dbl>      <dbl>
## 1 2005-01-01 163540.      100          42.3 103.      102.
## 2 2005-02-01 160702.      98.3         42.7 99.1      98.9
## 3 2005-03-01 175469.      107.         43.1 112.      98.3
## 4 2005-04-01 177179      108.         42.5 108.      107.
## 5 2005-05-01 177497.      109.         42.4 113.      100.
## 6 2005-06-01 180882.      111.         42.8 115.      104.
## 7 2005-07-01 184074.      113.         43.2 111.      99.8
## 8 2005-08-01 187247.      114.         43.2 120.      97
## 9 2005-09-01 181539.      111.         43.2 115.      96.1
## 10 2005-10-01 189183      116.         43.6 110.      95.8
## # ... with 174 more rows
```

Na seção 8.3, expliquei que ao preencher os argumentos já no `ggplot()` você estaria pedindo ao programa, que utilize a mesma base de dados e/ou o mesmo *aesthetic mapping* ao longo de todas as camadas do gráfico. Como exemplo, veja o que acontece no gráfico abaixo.

Como o `geom_bar()` busca resumir os nossos dados em poucas estatísticas, eu coloquei dessa vez o valor “*identity*” no argumento `stat`. Isso impede que ele agrupe os dados em alguma medida estatística, fazendo com que o `geom` apenas identifique os valores que aparecem na base, da mesma forma que um `geom_point()` faria. Perceba também, que eu estou utilizando três `geom`’s diferentes no gráfico. Mas como eu não defini um *aesthetic mapping* específico em cada um deles, todos esses `geom`’s estão mostrando exatamente a mesma informação. Dito de outra forma, estes `geom`’s estão utilizando o mesmo *aesthetic mapping*, o qual definimos na função `ggplot()`.

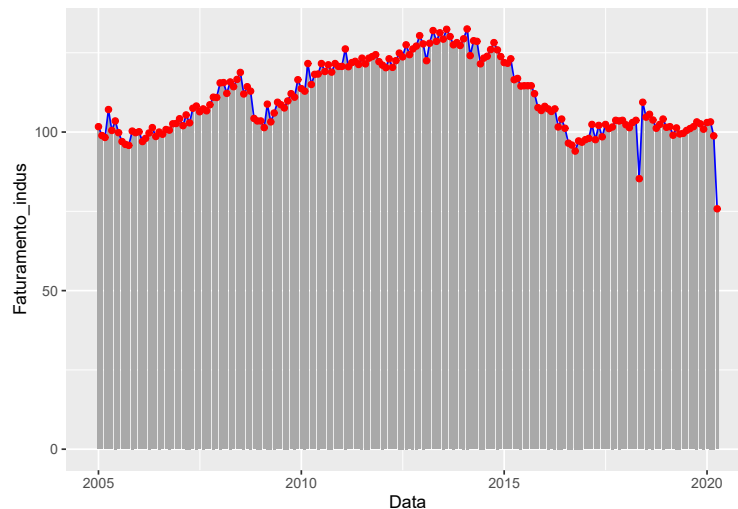
```
ggplot(
  data = PIB,
```

⁶<http://www.ipeadata.gov.br/Default.aspx>

```

aes(x = Data, y = Faturamento_indus)
) +
geom_bar(stat = "identity", fill = "darkgray") +
geom_line(color = "blue") +
geom_point(color = "red")

```



Fonte: Elaboração própria do autor.

Você deve estar pensando: “Ok, mas isso não faz sentido! Por que eu usaria três geom’s diferentes para uma mesma informação?”. Bem, pode ser que você queira utilizar mais de um geom que mostre a mesma informação, por questões estéticas no gráfico. Um exemplo simples, seria marcar a borda de uma linha criada por `geom_line()`. Ou seja, não há uma forma direta e simples em `geom_line()` (algo que já é possível de ser feito no `geom_point()`), de pintar essas bordas de uma cor mais escura (ou clara) do que o interior da linha, dando assim uma maior ênfase para aquela linha. Portanto, a ideia seria criarmos duas camadas de `geom_line()`: uma interna, com uma linha mais estreita e de cor mais clara (ou mais escura); e uma externa, com uma linha mais larga (de forma que ela “transborde” para fora da linha interna) e de cor mais escura (ou mais clara).

De qualquer maneira, esses geom’s não fazem muito sentido da forma como estão dispostos no momento, portanto, vamos mudar de estratégia. Por que não utilizamos um só geom para nos apresentar três informações diferentes?! Para isso, temos que modificar a nossa base levemente. O objetivo é pegar as três colunas com as variáveis que vamos plotar (`Faturamento_indus`, `FBCF` e `Divida_liq_perc`), e agrupá-las em duas colunas: uma com os valores dessas variáveis, e outra coluna com os nomes dessas variáveis, para identificar qual variável o valor na primeira coluna se refere. Realizamos esse trabalho pela função `pivot_longer()`.

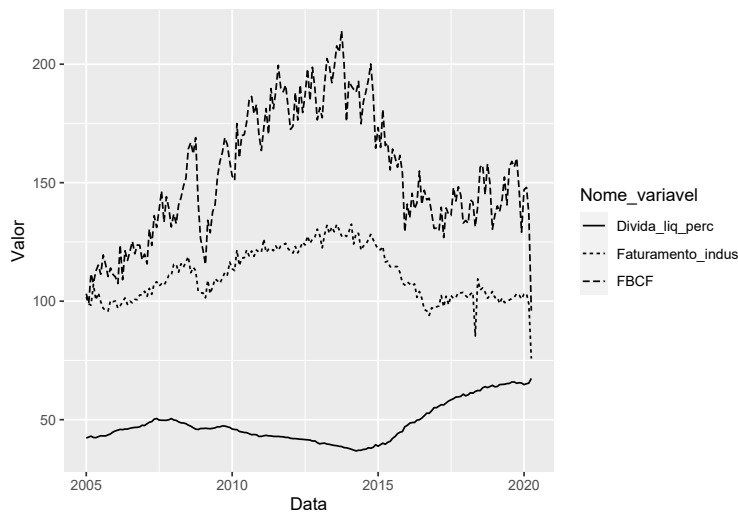
```

PIB_remodelado <- PIB %>%
  select(Data, Faturamento_indus, FBCF, Divida_liq_perc) %>%

```



```
pivot_longer(  
  cols = c("Faturamento_indus", "FBCF", "Divida_liq_perc"),  
  names_to = "Nome_variavel",  
  values_to = "Valor"  
)  
  
ggplot(  
  data = PIB_remodelado,  
  aes(x = Data, y = Valor, linetype = Nome_variavel)  
) +  
  geom_line()
```



Fonte: Elaboração própria do autor.

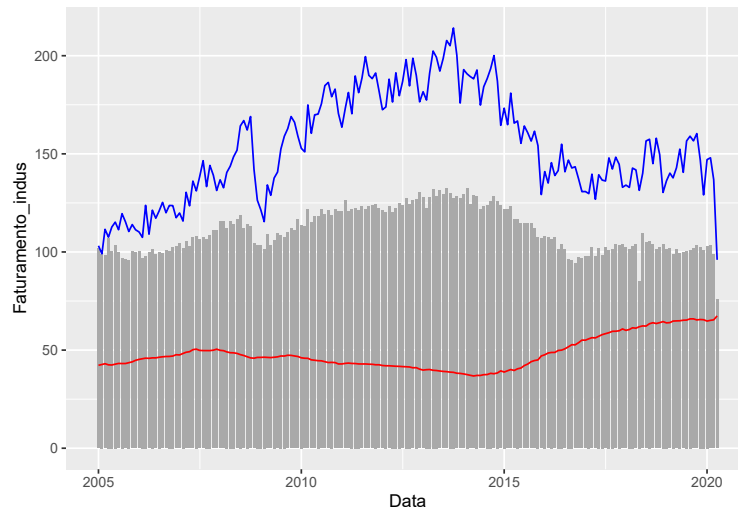
Novamente, como não incluímos uma função `aes()`, ou definimos o argumento `data` dentro do `geom_line()`, ele irá utilizar o `data` e o *aesthetic mapping* (`aes()`) que definimos em `ggplot()`. Lembra de quando eu disse que você poderia controlar o formato de um geom de acordo com uma variável? O gráfico acima é um exemplo prático disso. Estamos utilizando apenas um geom para mostrar três informações diferentes, e o componente estético que utilizamos para diferenciar essas informações no gráfico, é o formato dessas linhas. Portanto, ao definirmos o componente `linetype` para *Nome_variavel*, estamos modificando o formato da linha (tracejada ou sólida), de acordo com os valores dessa variável. Poderíamos usar a mesma estratégia em `geom_point()`, ao definirmos o argumento `shape` para *Nome_variavel*. O resultado, seria um gráfico com pontos de três formatos diferentes (triângulos, quadrados e pontos comuns).

Entretanto, para utilizarmos essa estratégia, nós tivemos que reestruturar a nossa base de dados pela função `pivot_longer()`. E se você não quisesse modificar essa base? Infelizmente, sem essa modificação, não poderíamos mostrar as três variáveis utilizando apenas uma função geom, mas poderíamos realizar o mesmo trabalho com uma função geom para cada variável. Neste caso,

teremos que utilizar um *aesthetic mapping* diferente para cada geom, pois cada um deles, ficará responsável por mostrar os valores de uma variável diferente.

No primeiro gráfico dessa seção, utilizamos três geom's diferentes para mostrar uma mesma informação. Se você comparar o código desse primeiro gráfico, ao código do gráfico abaixo, você perceberá que eles são quase idênticos, o que mudou, é a presença da função `aes()` nos dois últimos geom's.

```
ggplot(  
  data = PIB,  
  aes(x = Data, y = Faturamento_indus)  
) +  
  geom_bar(stat = "identity", fill = "darkgray") +  
  geom_line(aes(y = FBCF), color = "blue") +  
  geom_line(aes(y = Divida_liq_perc), color = "red")
```



Fonte: Elaboração própria do autor.

O único geom que não possui uma função `aes()` definida, é o `geom_bar()`, logo, esse geom vai seguir o *aesthetic mapping* que definimos em `ggplot()`. Já os outros dois geom's, vão seguir o *aesthetic mapping* que definimos em seus respectivos `aes()`. Porém, repare que em ambos geom's, eu apenas defini a variável mapeada para o eixo y, não cheguei a definir uma nova variável para o eixo x. Quando isso ocorre, a função irá novamente recorrer ao *aesthetic mapping* que você definiu em `ggplot()`. Ou seja, como não definimos uma nova variável para o eixo x, todos os geom's do gráfico acabam utilizando a variável no eixo x definida em `ggplot()`.

Portanto, você pode sobrepor por completo, ou parcialmente, o *aesthetic mapping* definido em `ggplot()` em cada geom, basta omitir os termos dos quais você não deseja sobrepor na nova função `aes()`. Um outro detalhe, é que não chegamos a definir em nenhum momento, um novo valor para o argumento `data` em algum geom. Logo, apesar de estarmos utilizando diferentes *aesthetic mappings*, todos os geom's estão utilizando a mesma base de dados.

8.5.1 Resumo da estrutura básica de um gráfico `ggplot()`

Em resumo, todo gráfico do `ggplot()` possui três camadas essenciais, que formam a base do gráfico: 1) *data*, a base (ou bases) de dados utilizada no gráfico em questão; 2) *aesthetic mapping*, o mapeamento, ou a ligação de variáveis presentes na base de dados, para componentes estéticos e visuais do gráfico; 3) *geom*, a forma geométrica (retângulos, pontos, polígonos, linhas, etc) que irá representar os seus dados no gráfico.

Para construir um gráfico do `ggplot()`, você deve sempre definir esses componentes. Os dois primeiros (*data* e *aesthetic mapping*), podem ser definidas dentro da função `ggplot()`, já o terceiro (*geom*), você define ao utilizar uma (ou várias) das funções *geom*, em uma (ou em várias) das camadas do gráfico. Com isso, temos uma estrutura básica como a definida abaixo, para construirmos um gráfico do `ggplot`:

```
ggplot(  
  data = <sua base de dados>,  
  aes(<aesthetic mapping>)  
) +  
  <geom_...> #uma função geom a seu gosto
```

Lembre-se que essa é apenas uma estrutura básica. Como mostramos na seção 8.4, podemos sobrepôr de diversas formas essa estrutura. E podemos definir diversos outros parâmetros sobre essa estrutura como foi mostrado ao longo do capítulo.

8.6 Uma discussão sobre os principais geom's

Nas próximas seções vamos descrever rapidamente como grande parte dos principais *geom's* se comportam, e quais são os argumentos (ou os componentes estéticos) que podemos controlar através da função `aes()`. Dessa forma, você pode rapidamente se familiarizar com esses *geom's*, adquirindo um vocabulário das funções que os representam, e que cobrem a maior parte dos gráficos realizados no dia-a-dia.

Lembre-se que há várias funções *geom* diferentes, das quais muitas não serão descritas aqui. Muitas deles utilizam o mesmo formato geométrico (linhas, retângulos, ...), mas desenham esse formato de uma maneira diferente, além de possuírem outros componentes estéticos que podem ser controlados pelo *aesthetic mapping* do gráfico.

Caso você não encontre aqui, o formato geométrico que está procurando, ou a função *geom* que realiza o desenho da forma como você deseja, você pode consultar a página oficial do pacote⁷. A página não possui uma versão em português, porém, você deve se virar razoavelmente bem com ferramentas de tradução (como o Google Tradutor) nessas situações. Se isso não for suficiente, você talvez encontre suas dúvidas em outros materiais construídos por brasileiros, como o blog *Curso*

⁷[<https://ggplot2.tidyverse.org/reference/index.html>](https://ggplot2.tidyverse.org/reference/index.html)

*R*⁸, o material do departamento de Estatística da UFPR⁹, ou dentro de alguma pergunta postada na página em português do *StackOverflow*¹⁰. O *R Graph Gallery*¹¹ é um repositório (em inglês) que possui exemplos dos mais diversos geom's, e que serve como a referência perfeita para os momentos em que você não lembra qual o geom que desenha o tipo de gráfico que procura.

8.6.1 Gráficos de dispersão e gráficos de bolha

Gráficos de dispersão são formados por `geom_point()`. Esse geom (por padrão) não transforma os seus dados, ou em outras palavras, ele não busca resumí-los de alguma maneira. Cada ponto desenhado no plano cartesiano representa cada uma das linhas presentes em sua base de dados. Os geoms que possuem este comportamento, são comumente chamados de geom's individuais. Por este padrão, você deve obrigatoriamente definir as variáveis de ambos os eixos (x e y), neste geom.

Nos exemplos abaixo, estou utilizando a tabela `mpg` que vêm junto do `ggplot`, e nos apresenta dados de consumo de combustível de diversos modelos de carro, para mais detalhes desses dados, execute `?mpg` no console. Os gráficos nos mostram uma relação aparentemente negativa entre volume ocupado pelos pistões no motor (`displ`), e a quilometragem por litro de gasolina (`hwy`).

Após definir os eixos, você pode pintar os pontos de acordo com uma terceira variável, por exemplo, a classe do carro (compacto, SUV, minivan, ...), através do argumento `color`. A classe do carro é uma variável categórica, e por isso, o `ggplot()` irá buscar cores contranstantes para pintar os pontos. Mas você também pode definir uma variável contínua a este argumento, onde neste caso, o `ggplot()` irá criar um gradiente de cores para pintar os pontos. Uma outra possibilidade deste geom, é variar o formato dos pontos através do argumento `shape`.

A partir de `geom_point()` você também pode construir um gráfico de bolha, através do argumento `size`. Este tipo de gráfico em geral, piora o *overplotting*, ou a sobreposição dos pontos, já que alguns ficam muito grandes. Nestas situações, o argumento `alpha` é bem útil, sendo ele definido por um número de 0 a 1, indicando uma porcentagem de opacidade do geom. Por padrão, ele é setado para 1 (100%), já no exemplo, eu reduzo essa opacidade para 40%.

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

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

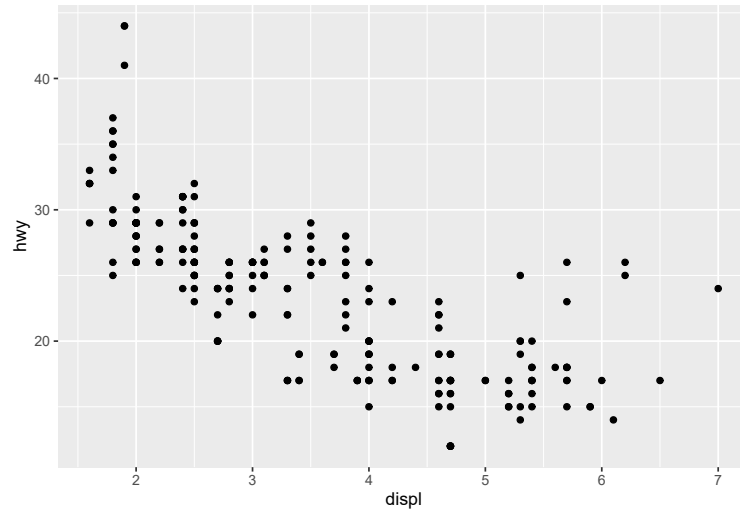
```
ggplot(data = mpg) +  
  geom_point(aes(x = displ, y = hwy, color = class, shape = drv))
```

⁸<https://www.curso-r.com/material/ggplot/>

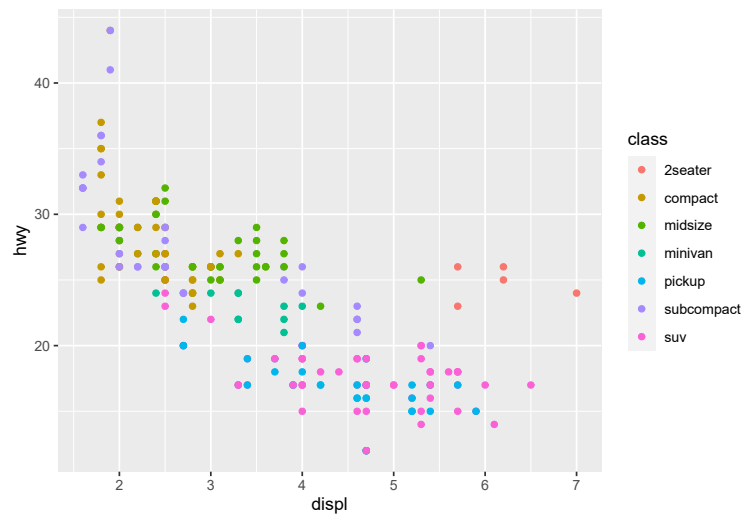
⁹<http://www.leg.ufpr.br/~walmes/cursor/data-vis/index.html>

¹⁰<https://pt.stackoverflow.com/>

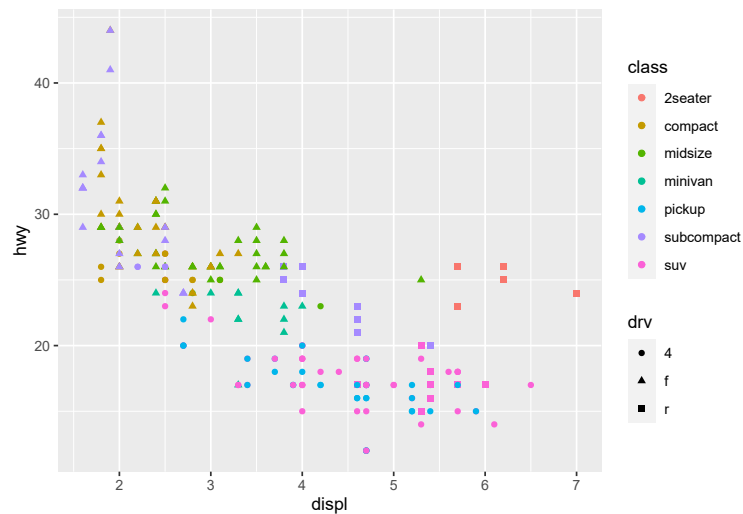
¹¹<https://www.r-graph-gallery.com/>



Fonte: Elaboração própria do autor.

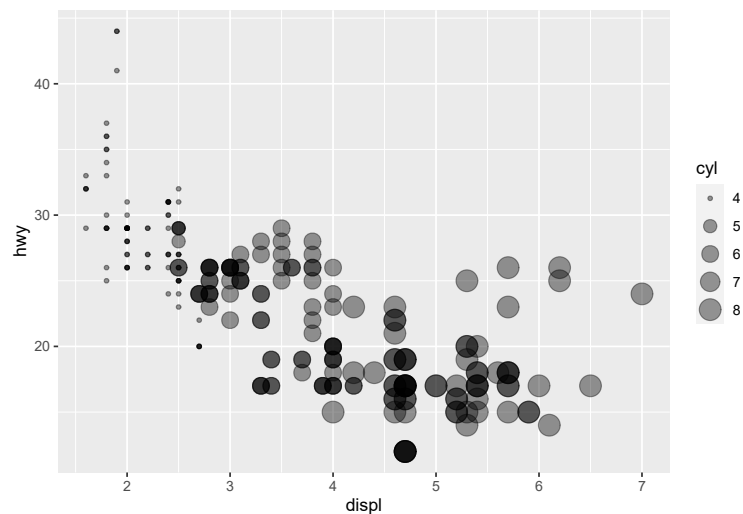


Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

```
ggplot(data = mpg) +  
  geom_point(aes(x = displ, y = hwy, size = cyl), alpha = 0.4)
```

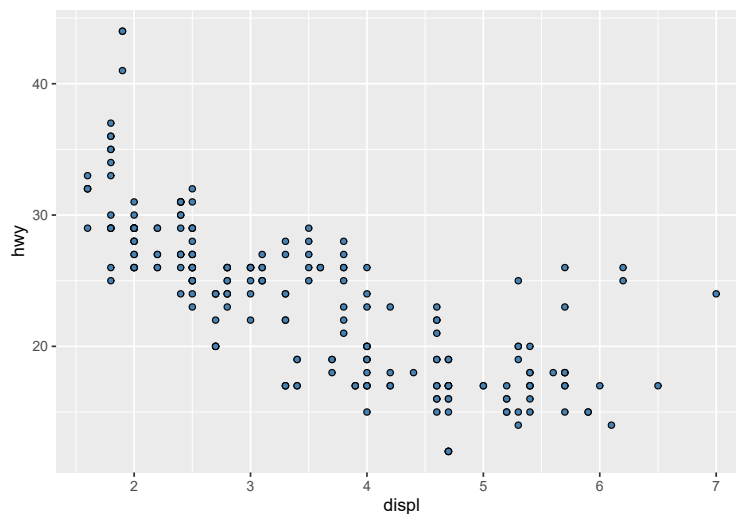


Fonte: Elaboração própria do autor.

Você pode se aproveitar do componente *shape* para diferenciar, ou destacar as bordas dos pontos, ao escolher o *shape* 21. Este método é esteticamente atraente, e fica muito bom em conjunto com linhas. Dessa forma, você pode pintar o interior dos pontos de uma cor, utilizando *fill*, e a borda desse ponto de outra cor, utilizando *color*. Lembre-se que isso só é possível, pelo *shape* que escolhemos para estes pontos, em outras situações, você poderá colorir pontos apenas com uma cor,

utilizando o *color*. No exemplo abaixo, eu deixo todos os três argumentos de fora de `aes()`, dessa forma, o `ggplot` mantém os valores que dei a cada um deles, constantes ao longo de todo o gráfico.

```
ggplot(data = mpg) +  
  geom_point(  
    aes(x = displ, y = hwy),  
    shape = 21,  
    color = "black",  
    fill = "steelblue"  
  )
```



Fonte: Elaboração própria do autor.

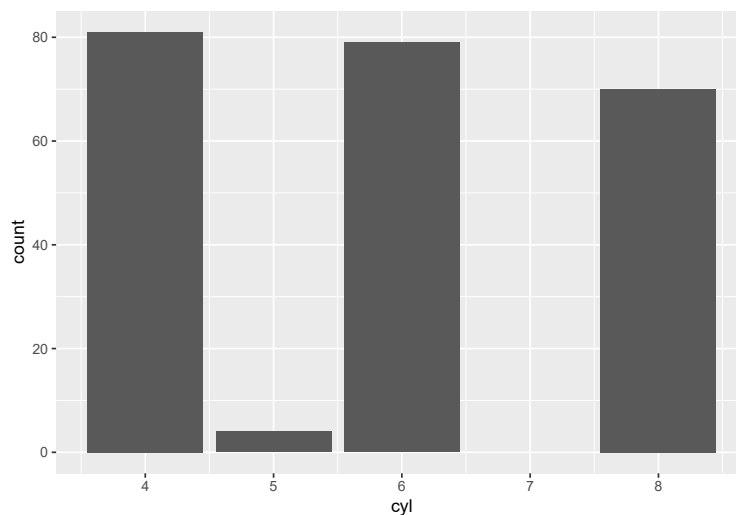
8.6.2 Gráficos de barra

Como descrevi anteriormente, os gráficos de barras no `ggplot()` são formados pelo `geom_bar()`, e em geral, são utilizados para apresentar estatísticas que resumem os dados em poucos números (como totais, médias, medianas). Em outras palavras, os geom's que tem este comportamento, buscam representar várias observações de sua base, com um único formato geométrico, e são comumente chamados de geom's coletivos. Por essa razão, o argumento *stat* é importante neste geom, pois nele você pode conceder o valor *identity*, que evita este comportamento, e faz com que o geom apenas identifique os valores que você fornece a ele.

No `ggplot`, este geom foi criado com o intuito de permitir que o usuário construa rapidamente gráficos de contagens e somatórios. Portanto, este geom possui mecanismos para calcular essas estatísticas, você não precisa calculá-las por conta própria antes de gerar o `ggplot`. Por padrão, este geom calcula inicialmente uma contagem dos seus dados. Logo, caso você não defina qual a estatística que deseja mostrar, ele irá contar a quantidade que cada valor aparece na base. Por

exemplo, o gráfico abaixo nos mostra que dentro da tabela `mpg`, temos em torno de 80 modelos com motores de 4 ou 6 cilindradas, e menos de 10 modelos com 5 cilindradas.

```
ggplot(  
  data = mpg,  
  aes(x = cyl)  
) +  
geom_bar()
```



Fonte: Elaboração própria do autor.

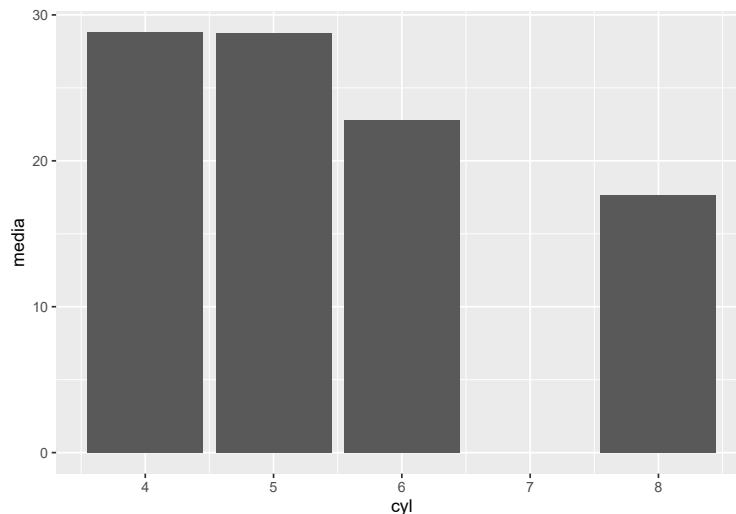
Tendo essas considerações em mente, você tem duas opções básicas ao lidar com este geom: 1) fornecer diretamente os dados, e pedir ao geom que calcule as estatísticas que você deseja mostrar (contagem ou somatório); ou 2) você primeiro calcula as estatísticas que deseja, e pede ao geom que apenas as identifique, sem realizar nenhuma transformação desses dados. Caso opte pela opção 2, você deve **tomar muito cuidado** com o argumento `stat = "identity"`, por razões que vou explicar abaixo.

Este geom não possui um mecanismo próprio para calcular médias (e muitas outras estatísticas), e portanto, se você quiser mostrá-las utilizando este geom, você terá de calcular separadamente essas médias, e pedir ao geom que apenas as identifique com `stat = "identity"`.

```
medias <- mpg %>%  
  group_by(cyl) %>%  
  summarise(media = mean(hwy))  
  
ggplot(  
  data = medias,  
  aes(x = cyl, y = media)
```



```
) +  
geom_bar(stat = "identity")
```



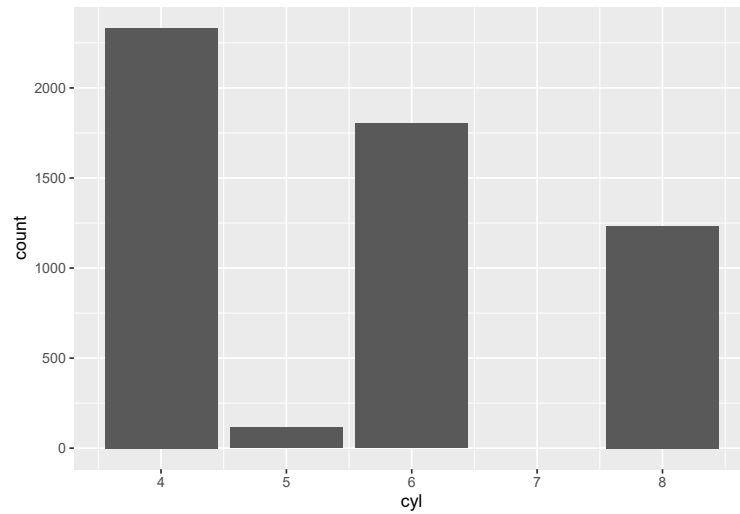
Fonte: Elaboração própria do autor.

Porém, caso você queira calcular o total, ou o somatório em cada grupo, você pode apenas definir a coluna com os valores a serem somados, para o argumento *weight* dentro de *aes()*.

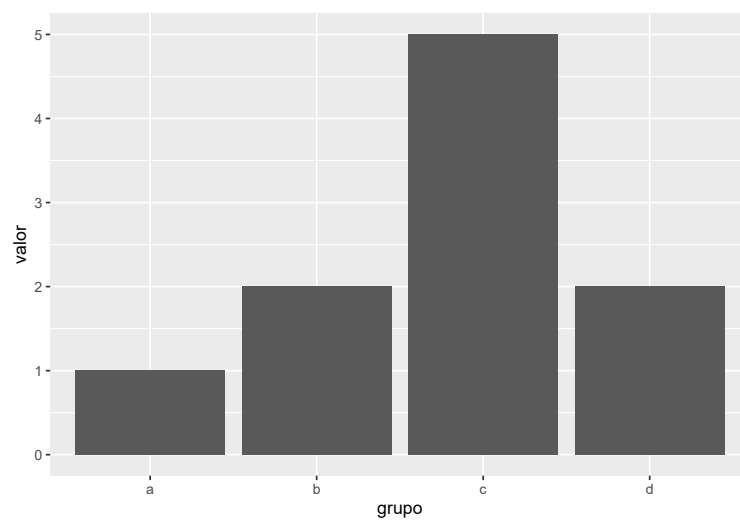
```
ggplot(  
  data = mpg,  
  aes(x = cyl, weight = hwy)  
) +  
geom_bar()
```

Agora, lembra quando eu disse que você pode pedir ao geom que apenas “identifique” os valores de sua base (com *stat = "identity"*)? Com este argumento, o *geom_bar()* irá ter um comportamento diferente, caso os valores em cada grupo não sejam únicos. No exemplo anterior, em que calculei as médias de cada *cyl* em *mpg*, o geom apenas identificou as médias de cada *cyl*, pois há apenas **uma única** média para cada *cyl*. No exemplo abaixo, estou criando rapidamente uma tabela, e nela você pode perceber que há dois valores para o grupo "c". Agora, repare o que acontece no gráfico, o *geom_bar()* acaba somando estes valores.

```
tab <- data.frame(  
  grupo = c("a", "b", "c", "c", "d"),  
  valor = c(1, 2, 3, 2, 2)  
)  
  
ggplot(tab, aes(x = grupo, y = valor)) +  
  geom_bar(stat = "identity")
```



Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

Em outras palavras, se os seus dados estiverem **agrupados**, o `geom_bar()` com `stat = "identity"` irá de fato apenas identificar estes valores. Mas caso os seus dados estiverem desagregados, com mais de um valor por grupo, o `geom_bar()` irá somar estes valores. Isso significa, que `stat = "identity"` representa uma outra alternativa (além de *weight*), para criar gráficos de somatório. Bastaria fornecer a coluna com os valores a serem somados para o eixo y em `aes()`, e adicionar `stat = "identity"` à `geom_bar()`.

Um outro ponto importante neste geom, é o posicionamento das barras. Por padrão, o geom empilha barras que ocupam o mesmo valor no eixo x no gráfico. Isso nos permite visualizarmos a participação dos grupos de uma outra variável categórica (cor de pele, faixa etária, ...), em cada valor presente no eixo x. Por outro lado, você talvez esteja interessado na diferença entre os grupos, e não a sua participação. Logo, você talvez queira jogar essas barras uma do lado da outra, e para isso você deve utilizar o argumento *position*, dando o valor “*dodge*”. No exemplo abaixo, retorno a base de dados `datasus`, com o objetivo de mostrar a diferença em cada cor de pele, da média de vítimas para cada sexo.

```
medias <- datasus %>%
  group_by(Cor, Genero) %>%
  summarise(media = mean(Contagem))

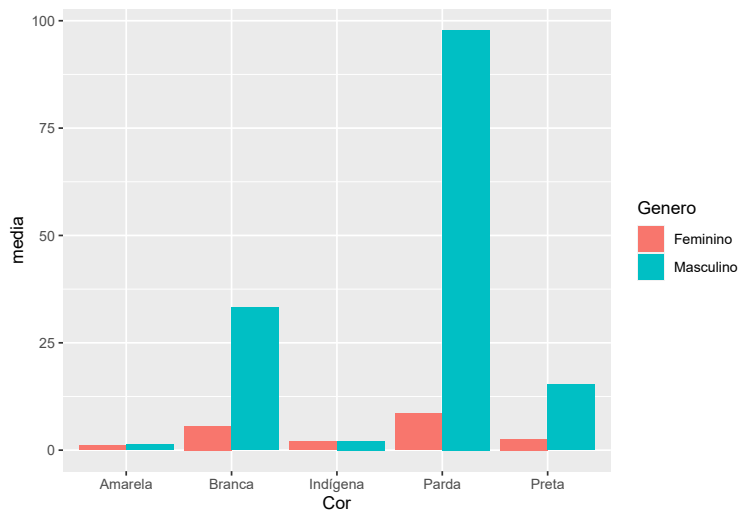
## `summarise()` has grouped output by 'Cor'. You can override using the `.groups` argument.
```

```
ggplot(
  data = medias,
  aes(x = Cor, y = media, fill = Genero)
) +
  geom_bar(stat = "identity", position = "dodge")
```

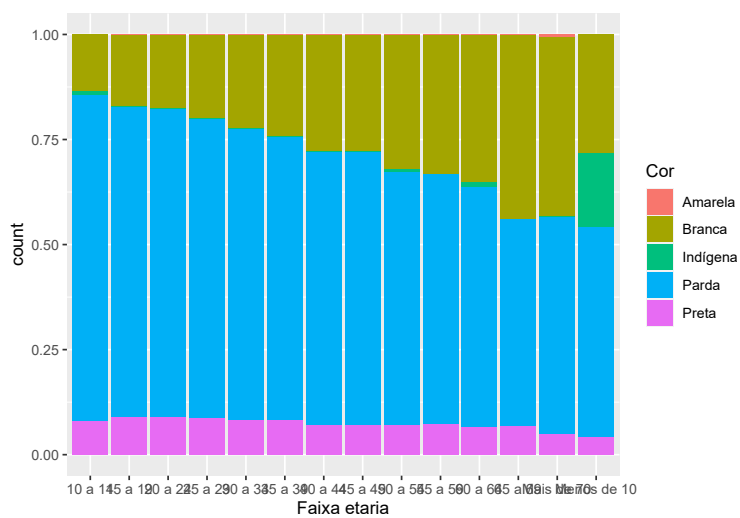
Portanto, caso você não definisse *position* para *dodge*, o `ggplot` iria empilhar essas barras (azul e vermelho) uma em cima da outra. Em um gráfico de médias como o acima, não faria muito sentido empilhar essas barras, porém, este posicionamento faz muito sentido em gráficos de somatório, como os que fizemos na seção 8.3. Pois dessa forma você consegue visualizar o quanto que cada grupo representa do total.

Você talvez queira ir um pouco além, e observar as diferenças na participação de cada cor de pele, ao longo dos totais de vários grupos. Para isso, você pode dar o valor *fill* ao argumento *position*. Dessa forma, o `ggplot` calcula qual é a proporção de cada grupo para cada valor do eixo x, em uma escala de 0 a 1. Nesta situação, você deve definir a variável do eixo y, para o argumento *weight* em `aes()`.

```
ggplot(
  data = datasus,
  aes(x = `Faixa etaria`, weight = Contagem, fill = Cor)
) +
  geom_bar(position = "fill")
```



Fonte: Elaboração própria do autor.



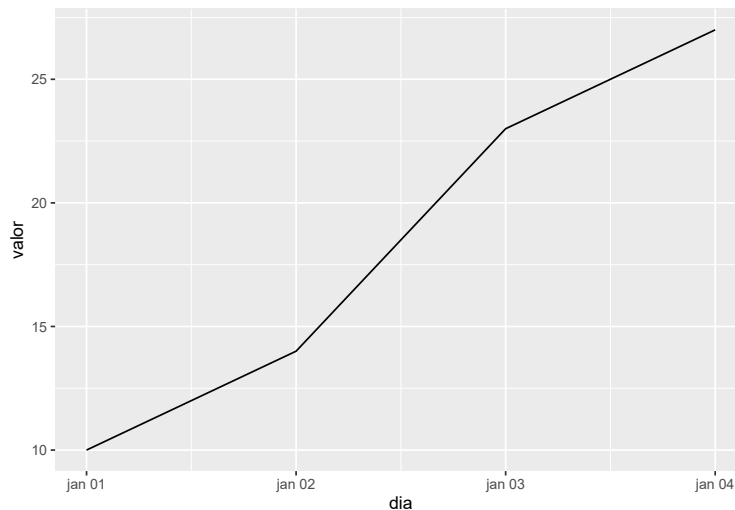
Fonte: Elaboração própria do autor.

8.6.3 Gráficos de linha

Gráficos de linha são muito utilizados em séries temporais, para mostrar a evolução de algum índice ao longo do tempo. Este tipo de gráfico é criado pelo `geom_line()`, que possui um comportamento de “conector”.

O `geom_line()` (diferentemente de seu irmão `geom_path()`) sempre ordena os valores da base (antes de conectá-los), segundo a variável alocada no eixo x, na ordem que seja mais lógica para essa variável. Veja o exemplo abaixo, onde dou um vetor de datas (fora de ordem) para o eixo x. Independente da ordem em que os valores estiverem em sua base, a função irá reordenar a base antes de conectar os pontos.

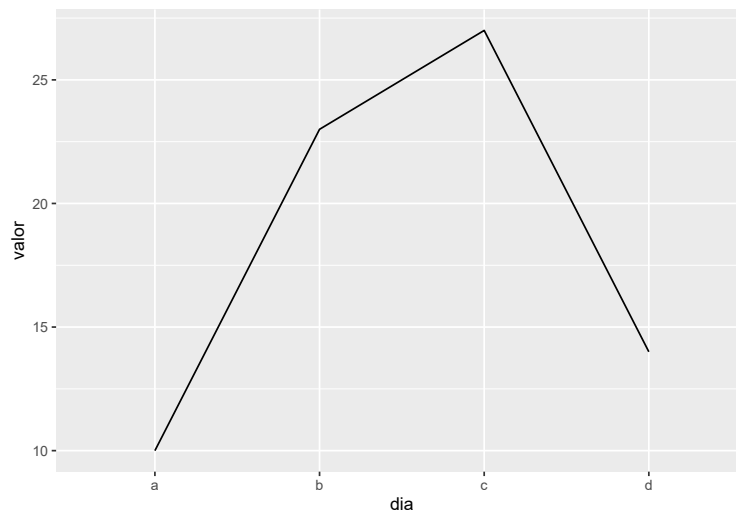
```
tab <- data.frame(  
  dia = as.Date(c("2020-01-01", "2020-01-04", "2020-01-02", "2020-01-03")),  
  valor = c(10, 27, 14, 23)  
)  
  
ggplot(tab, aes(dia, valor)) + geom_line()
```



Fonte: Elaboração própria do autor.

Isso significa, que este geom funciona com qualquer variável no eixo x que possua uma ordem lógica, seja ela contínua ou categórica. Veja no exemplo abaixo, onde eu substituo a coluna dia, por um simples vetor de texto. Ao detectar o tipo de dado presente na coluna, a função reordena os valores de acordo com a ordem lógica para este tipo de dado (no exemplo abaixo, ordem alfabética).

```
tab$dia <- c("a", "c", "d", "b")  
  
ggplot(tab, aes(dia, valor, group = 1)) + geom_line()
```



Fonte: Elaboração própria do autor.

Conhecer essa funcionalidade é importante, ao fornecer para o geom dados dos quais ele não consegue reconhecer o formato e a ordem correta. Pense por exemplo, se você fornecesse uma vetor de datas, mas no formato “Abril/2020”. Como os valores começam por um nome, ele reconhece estes valores como texto, e portanto, ordena-os em ordem alfabética, ao invés de ordená-los como meses do ano. Nessas situações, transformar esses valores para fatores, e definir a sua ordem em seu atributo `levels`, é a melhor alternativa.

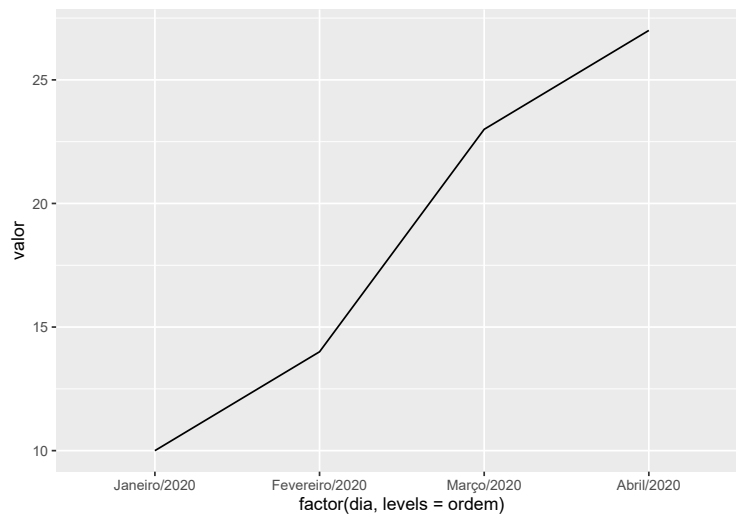
```
tab$dia <- c("Janeiro/2020", "Abril/2020", "Fevereiro/2020", "Março/2020")
```

```
ordem <- c("Janeiro/2020", "Fevereiro/2020", "Março/2020", "Abril/2020")
```

```
ggplot(  
  tab,  
  aes(factor(dia, levels = ordem), valor, group = 1)  
) +  
  geom_line()
```

Eu costumo aumentar a grossura dessas linhas através do argumento `size`, que por padrão está setado para 0.5. Geralmente 1 já é um bom nível para mim, mas você pode aumentar o quanto quiser. Como eu quero que a grossura, permaneça constante ao longo de toda a linha, eu mantenho o argumento `size` de fora do `aes()`. Isso significa que você poderia variar essa grossura ao longo da linha, apesar de que o resultado seria um tanto esquisito. Tente por exemplo, adicionar ao `aes()` do exemplo anterior, o valor `size = valor`, e veja o resultado.

Neste geom, o argumento `group` em `aes()` é muito importante. Este argumento controla como o geom considera os grupos da base, na hora de desenhar o formato geométrico em questão. No primeiro exemplo dessa seção, nós não utilizamos este argumento, pois a variável ligado ao eixo x



Fonte: Elaboração própria do autor.

(dia) era uma variável contínua. Entretanto, no instante em que mudamos os valores dessa coluna para texto, tivemos que adicionar um `group = 1` ao `aes()`. Logo, quando você ligar uma variável contínua ao eixo x, muito provavelmente você não precisará mexer com o `group`. Caso a variável seja categórica, é certo que algum valor deve ser dado ao argumento `group`.

Isso é apenas uma simplificação, que serve como um guia inicial, mas que nem sempre se aplica. Pois o `group` não diz respeito ao tipo de variável (contínua ou categórica), e sim se você quer separar ou não as linhas por algum agrupamento. Se você está apenas querendo mostrar uma única linha no gráfico, essa simplificação será útil. Mas com o tempo você vai se pegar utilizando o `group`, para mostrar em um mesmo gráfico a evolução de vários índices diferentes ao longo do tempo, mesmo que a variável no eixo x (datas) seja uma variável contínua. Basta lembrar o exemplo da seção 8.4, em que utilizamos `linetype` para diferenciar as curvas de três indicadores diferentes em um mesmo `geom_line()`. Você poderia replicar o mesmo gráfico utilizando `group`, ao invés do `linetype`.

Essa questão fica mais clara, ao utilizarmos uma base que possui mais de um valor por grupo. Veja por exemplo a base `Oxboys`, que vem do pacote `mlmRev`. Essa base é resultado de uma pesquisa, onde os pesquisadores acompanharam durante vários anos, o crescimento de alguns indivíduos.

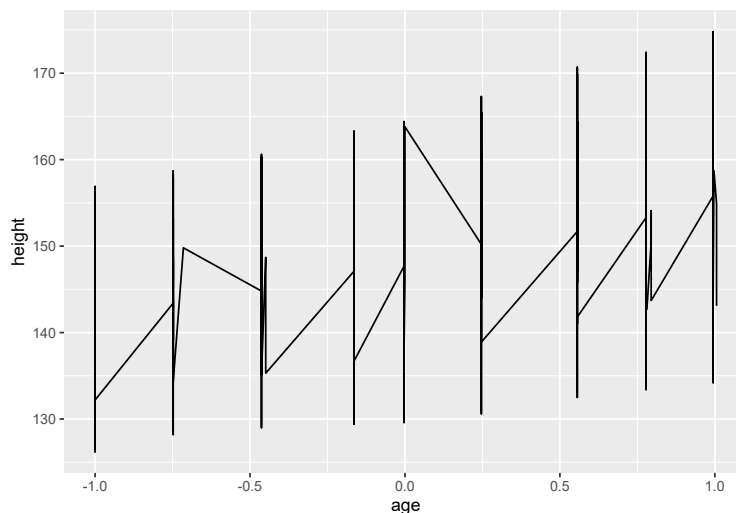
```
head(mlmRev::Oxboys, n = 10)
```

##	Subject	age	height	Occasion
## 1	1	-1.0000	140.5	1
## 2	1	-0.7479	143.4	2
## 3	1	-0.4630	144.8	3
## 4	1	-0.1643	147.1	4
## 5	1	-0.0027	147.7	5
## 6	1	0.2466	150.2	6

```
## 7      1  0.5562 151.7      7
## 8      1  0.7781 153.3      8
## 9      1  0.9945 155.8      9
## 10     2 -1.0000 136.9      1
```

Portanto, a coluna Subject identifica qual o indivíduo os valores da linha se referem. Repare que várias linhas dizem respeito ao mesmo indivíduo. Agora, pense como o `geom_line()` trataria esses diversos valores que se encaixam no mesmo grupo (no nosso caso, no mesmo Subject). Neste caso, o `geom_line()` irá conectar (incorretamente) todos os valores em conjunto da base, pois ele não sabe que cada um desses valores pertencem a sujeitos diferentes, o geom pensa que todos esses valores pertencem a um único sujeito. O resultado seria um gráfico com um aspecto de “serra”.

```
ggplot(
  Oxboys,
  aes(x = age, y = height)
) +
  geom_line()
```



Fonte: Elaboração própria do autor.

Para que isso fique claro, eu adicionei um `geom_point()` para que você veja cada um dos valores presentes na base. Primeiro, preste atenção nas variáveis que conectamos aos eixos do gráfico (idade e altura do indivíduo). Ambas as variáveis são contínuas, mas neste momento, não há qualquer variável no gráfico que possa identificar a qual dos indivíduos, cada um desses valores se refere. Logo, o `geom_line()` acaba conectando todos esses pontos juntos.

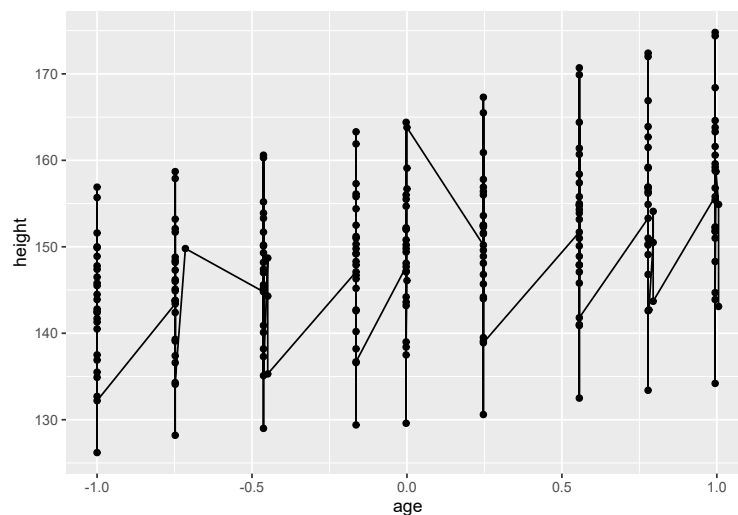
```
ggplot(
  Oxboys,
```



```

aes(x = age, y = height)
) +
geom_line() +
geom_point()

```



Fonte: Elaboração própria do autor.

Ao invés do `geom_line()` conectar todos esses pontos em conjunto, o geom deveria conectar todos os pontos que dizem respeito ao mesmo indivíduo, e é para isso que o argumento `group` serve. Você define neste argumento, qual a coluna que identifica qual é o grupo (ou no nosso caso, o indivíduo) que está sendo tratado em cada observação de sua base de dados.

```

ggplot(
  Oxboys,
  aes(x = age, y = height, group = Subject)
) +
  geom_line() +
  geom_point()

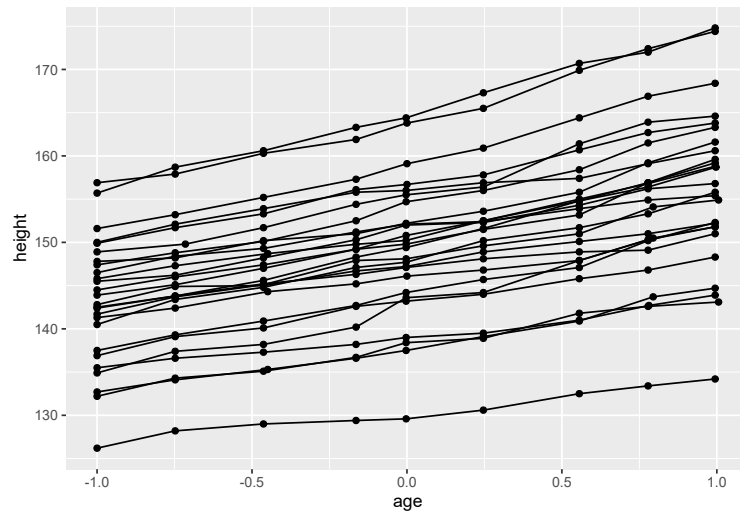
```

Uma outra forma de definirmos esses grupos para o geom, é colorindo as linhas com o argumento `color`, ou então variando o formato dessas linhas com o argumento `linetype`. Basta você fornecer a estes argumentos, uma coluna que seja capaz de identificar cada um dos grupos ou indivíduos (no nosso caso, `Subject`) que estão sendo tratados no gráfico.

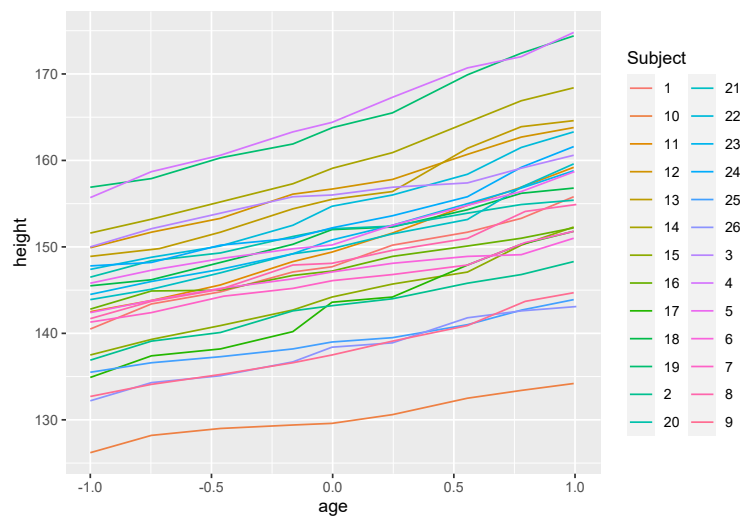
```

ggplot(
  Oxboys,
  aes(x = age, y = height, color = Subject)
) +
  geom_line()

```



Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

Portanto, toda vez em que utilizar este geom em uma base que possui mais de um valor por grupo, você muito provavelmente terá de utilizar *group*, especialmente se você precisa diferenciar as curvas de cada grupo no gráfico.

Se você quiser mostrar uma única linha no gráfico, você vai mexer obrigatoriamente com o *group* caso a variável do eixo x seja categórica, onde neste caso, você deve dar uma constante qualquer ao argumento (eu geralmente defino para 1: `aes(group = 1)`). Isso é necessário, porque `geom_line()` entende que cada um dos valores dessa variável categórica, representa um grupo diferente. Dessa forma, cada um desses grupos irá possuir apenas um valor em toda a base. Caso você se esqueça de definir este valor para *group* nesta situação, o seguinte erro irá aparecer:

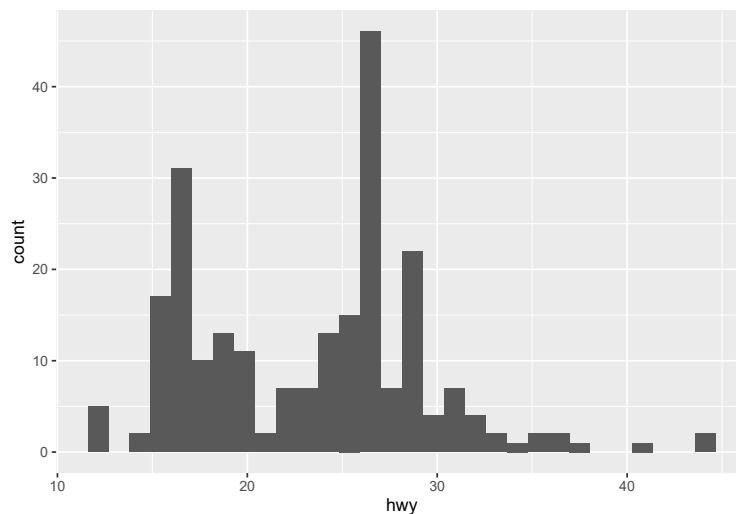
```
# Each group consists of only one observation. Do you need to adjust the group
# aesthetic?
```

8.6.4 Histogramas e outros gráficos de frequência

Histogramas e polígonos de frequência são gráficos “unidimensionais”, ou dito de outra forma, apresentam informações sobre apenas uma variável, mais especificamente uma variável contínua. Por essa razão, você precisa definir apenas um dos eixos do gráfico, geralmente, o eixo x. Estes gráficos são criados por `geom_histogram()` e `geom_freqpoly()`.

```
ggplot(mpg, aes(hwy)) + geom_histogram()
```

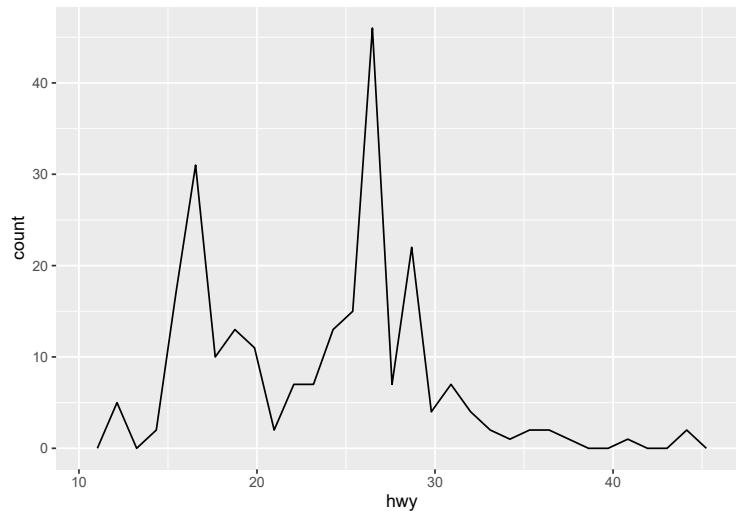
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Fonte: Elaboração própria do autor.

```
ggplot(mpg, aes(hwy)) + geom_freqpoly()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Fonte: Elaboração própria do autor.

Ambos os gráficos funcionam da mesma forma, apenas a forma geométrica utilizada é diferente. Eles pegam a distribuição da variável ligada ao eixo x, e dividem essa distribuição em vários intervalos (chamados de *bin*'s), e contam quantos valores se encaixam em cada um destes intervalos. Neste geom, é importante que você teste diferentes larguras para estes intervalos, através do argumento *binwidth*. Por padrão, o geom tenta dividir a distribuição em 30 intervalos diferentes.

Você pode separar as distribuições por alguma variável categórica, dando essa variável ao argumento *group*. Porém, essas distribuições estarão sobrepostas no gráfico, sendo impossível diferenciá-las. Logo, é necessário que você mostre essas distribuições separadas em diferentes facetas do gráfico (através da função *facet_wrap()*).

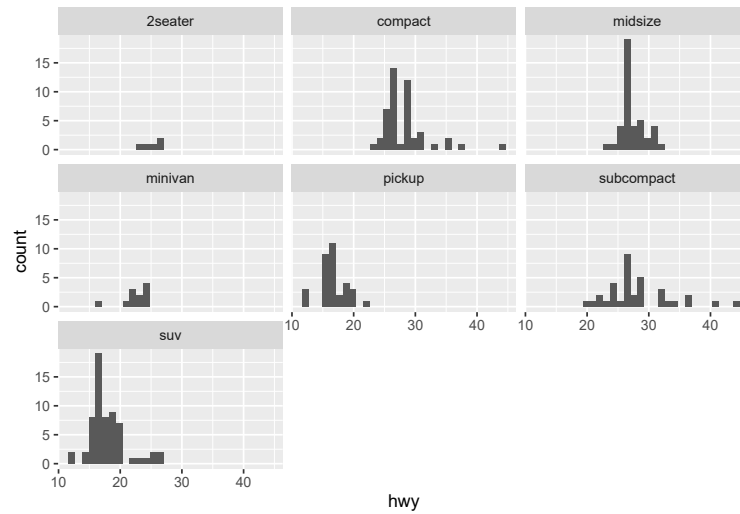
```
ggplot(mpg, aes(hwy, group = cyl)) +  
  geom_histogram() +  
  facet_wrap(~class)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

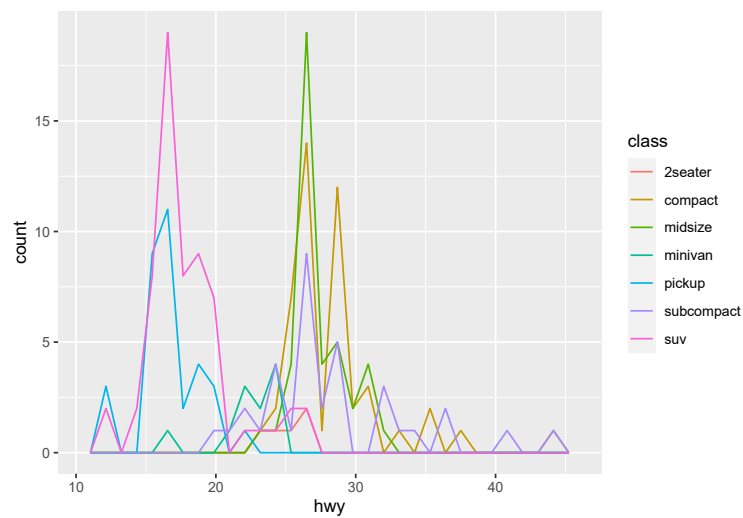
O *geom_freqpoly()* não sofre seriamente deste problema, pois a sua forma geométrica é “oca”. Mas é interessante de qualquer forma, que você ou separe essas distribuições em diferentes facetas do gráfico, ou então, que cora as distribuições de acordo com a variável categórica utilizada.

```
ggplot(mpg, aes(hwy, color = class)) +  
  geom_freqpoly()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



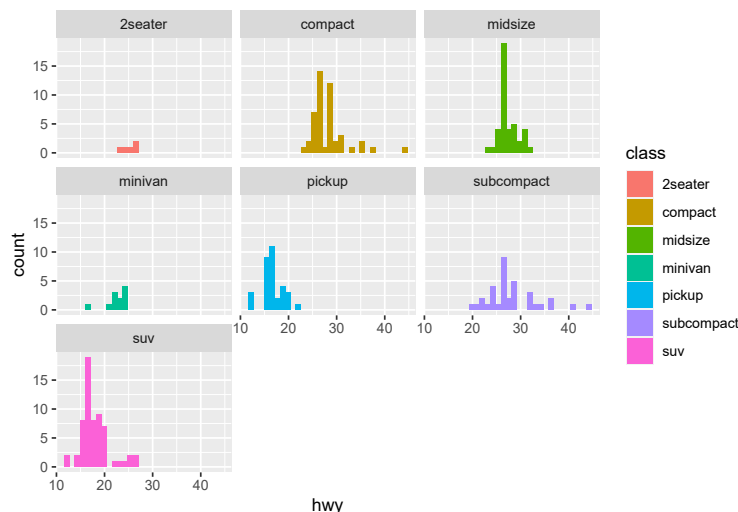
Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

```
ggplot(mpg, aes(hwy, fill = class)) +
  geom_histogram() +
  facet_wrap(~class)

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Fonte: Elaboração própria do autor.

Uma alternativa à estes geom's, é o `geom_density()` que calcula uma função de densidade para a variável escolhida. Caso esteja interessado em separar essa distribuição de acordo com uma variável categórica, eu recomendo que dê uma olhada no pacote `ggridges`. Este pacote fornece novos geom's, que posicionam essas distribuições separadas de uma forma esteticamente atraente, sem a necessidade de construir diferentes facetas do mesmo gráfico, além de fornecer mecanismos para marcar os percentis da distribuição no gráfico. É mais fácil ver com seus próprios olhos ¹², do que eu explicar.

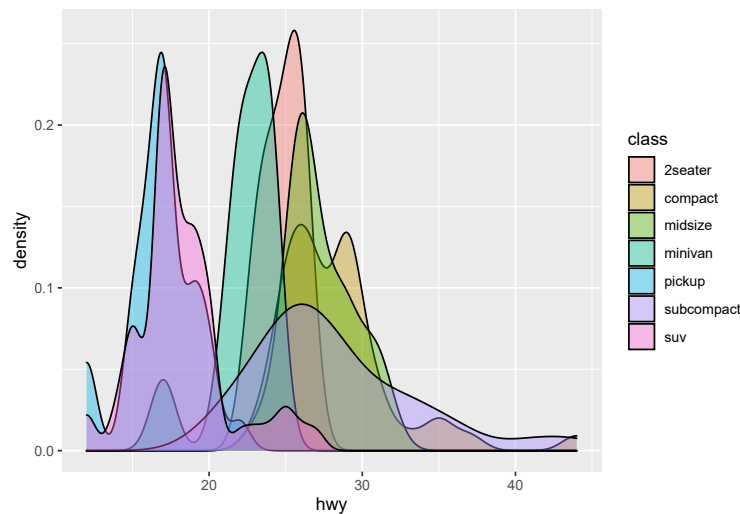
Caso você prefira permanecer com o geom padrão do `ggplot` e ainda separar a distribuição por uma variável categórica, você pode utilizar o argumento *alpha* para reduzir a opacidade dessas distribuições, como um meio de combater a sobreposição. Mas o ideal, é que você as separe em diferentes facetas, utilizando `facet_wrap()` da mesma forma que fizemos para os histogramas.

```
ggplot(mpg, aes(hwy, fill = class)) + geom_density(alpha = 0.4)
```

8.6.5 Adicionando textos ao gráfico

Você pode adicionar rótulos ao seu gráfico com `geom_label()`, ou adicionar textos simples com `geom_text()`. Estes geom's funcionam exatamente como o `geom_point()`, porém, ao invés de

¹²Veja a página oficial do pacote: <https://cran.r-project.org/web/packages/ggridges/vignettes/introduction.html>



Fonte: Elaboração própria do autor.

desenharem pontos, eles desenharam textos. Em outras palavras, eles são geom's individuais, em que desenharam um texto, ou um rótulo, para cada uma das observações de sua base de dados.

Dessa vez, você deve definir a coluna que contém os rótulos/textos que deseja mostrar no gráfico, no argumento *label* em *aes()*. Os rótulos serão posicionados no plano cartesiano de acordo com os valores definidos pelas variáveis ligadas aos eixos x e y.

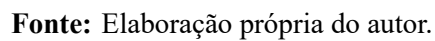
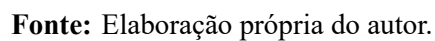
```
ggplot(mpg, aes(x = displ, y = hwy, label = model)) +  
  geom_text()
```

```
ggplot(mpg, aes(x = displ, y = hwy, label = model)) +  
  geom_label()
```

Ao colocar textos em um gráfico, você dificilmente não enfrentará algum nível de sobreposição. O ggplot oferece algumas ferramentas que em muitas ocasiões não resolvem o problema, mas que em outras podem ser suficientes. Ambos os geom's descritos aqui, possuem o argumento *check_overlap*. Caso ele seja configurado para TRUE, o ggplot irá criar os rótulos na ordem em que eles aparecem na sua base, e eliminar todos os rótulos consecutivos que sobrepõem os anteriores. O código ficaria dessa forma:

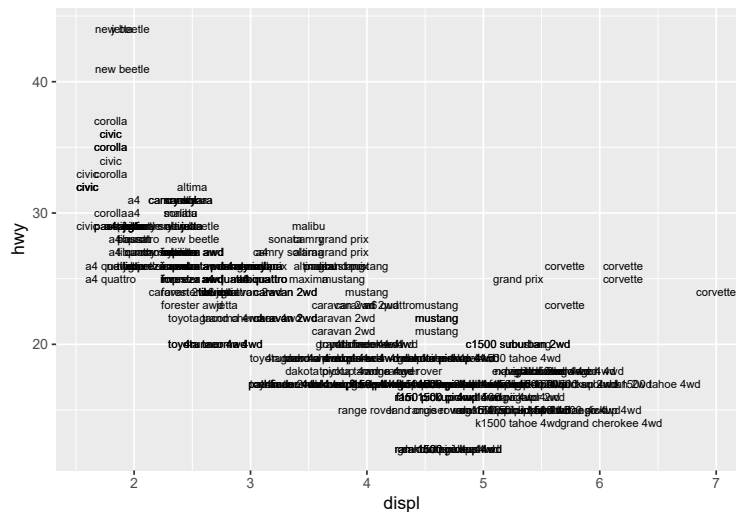
```
ggplot(mpg, aes(x = displ, y = hwy, label = model)) +  
  geom_text(check_overlap = TRUE)
```

Apesar de uma solução, ela pode muito bem eliminar justamente os rótulos que queremos destacar no gráfico, e por isso é pouco desejada. Você poderia também reduzir o tamanho da fonte através



de *size*. Um detalhe é que este argumento trabalha por padrão com valores em milímetros (mm), mas como é um pouco confuso trabalhar com tamanho de fontes nesta unidade, eu geralmente transformo os valores para pontos (pt). No exemplo abaixo, estou reduzindo o tamanho das fontes para 7 pt. O problema dessa opção, é que ela representa um *trade-off* entre a sobreposição de pontos, e a legibilidade dos rótulos, cabe a você definir o equilíbrio entre essas opções.

```
ggplot(mpg, aes(x = displ, y = hwy, label = model)) +  
  geom_text(size = 7/.pt)
```



Fonte: Elaboração própria do autor.

A melhor solução possível, seria ajustarmos a posição de cada um dos pontos individualmente. Entretanto, se você tem vários textos que exigem desvios diferentes, essa solução facilmente se torna muito trabalhosa. A ideia, seria criarmos duas novas colunas em nosso `data.frame`, onde em cada uma você define um valor de desvio vertical (`y_desvio`), e na outra o valor de desvio horizontal (`x_desvio`) para o rótulo definido naquela linha. Em seguida, você conecta essas colunas aos argumentos de posição responsáveis por realizar estes deslocamentos de textos (`nudge_y` e `nudge_x`) em seu *aesthetic mapping* (`aes()`). Veja o código abaixo.

```
ggplot(  
  mpg,  
  aes(  
    x = displ,  
    y = hwy,  
    label = model,  
    nudge_x = x_desvio,  
    nudge_y = y_desvio  
  )) +  
  geom_text()
```

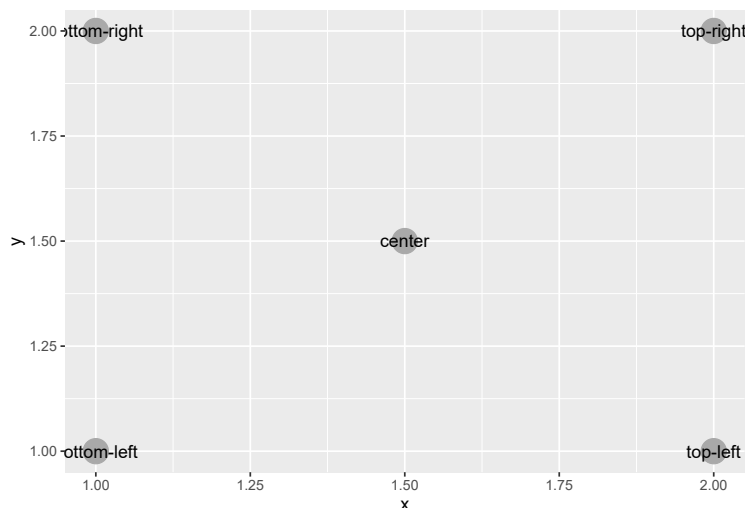
Vamos separar uma seção para descrevermos outras soluções mais eficazes para esse problema. Também vamos separar, uma seção para descrevermos quais são as estratégias possíveis para você trocar a fonte dos textos mostrados no gráfico, algo que ainda é difícil de ser realizado, especialmente se você trabalha no Windows. Agora vou explicar o que os argumentos de posição (*nudge_x* e *nudge_y*), e os de justificação (*hjust* e *vjust*) fazem.

Durante muito tempo, eu sofri de uma leve confusão entre esses argumentos. Como você muito provavelmente vai querer ajustar o posicionamento desses textos, vou tentar explicar a diferença entre os dois da forma mais clara possível, para que você não sofra do mesmo efeito.

Vamos começar pelos argumentos de justificação, que são *hjust* (justificação horizontal) e *vjust* (justificação vertical). Estes argumentos, servem para alterar a justificação, ou o alinhamento da cadeia de texto em relação ao seu ponto de referência (ou de coordenadas).

```
df <- data.frame(
  x = c(1, 1, 2, 2, 1.5),
  y = c(1, 2, 1, 2, 1.5),
  text = c(
    "bottom-left", "bottom-right",
    "top-left", "top-right", "center"
  )
)

ggplot(df, aes(x, y)) +
  geom_point(color = "darkgray", size = 7) +
  geom_text(aes(label = text))
```



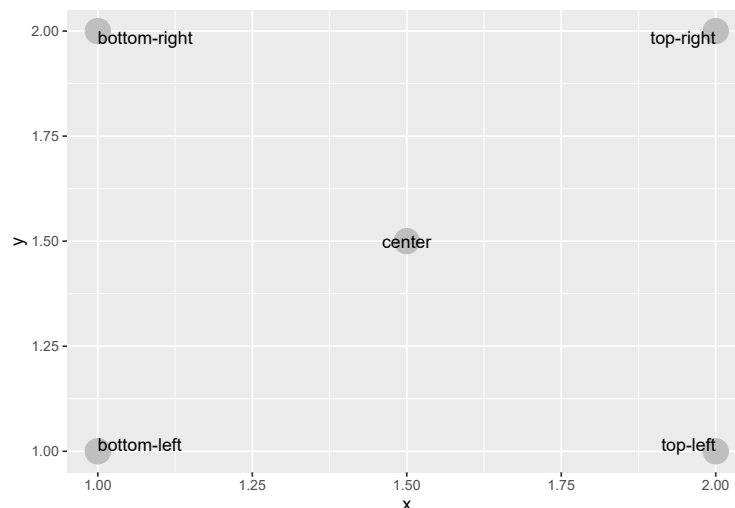
Fonte: Elaboração própria do autor.

Por padrão, *hjust* é setado para *center*, e *vjust* para *middle*. Logo, todos os rótulos são centralizados (tanto verticalmente, quanto horizontalmente) no ponto que define a sua localização. Para mudar o alinhamento de todos os rótulos de uma vez, você pode setar estes argumentos, por fora do `aes()`, fornecendo um dos valores pré-definidos.

No caso de *hjust*, há outros quatro valores pré-definidos possíveis (*left*, *right*, *inward*, *outward*). Caso você coloque *left* ou *right* neste argumento, todos os rótulos serão alinhados à esquerda, ou à direita dos pontos. Porém, caso você coloque *inward* ou *outward*, os textos serão alinhados (horizontalmente em relação aos pontos de sua localização) em um sentido para o para o centro do gráfico, ou se afastando do centro do gráfico. Dito de outra forma, os textos serão alinhados à esquerda, ou à direita do ponto de referência, a depender da sua localização em relação ao centro do plano cartesiano e do sentido escolhido (*inward* ou *outward*).

Para *vjust*, há também quatro outros valores pré-definidos (*bottom*, *top*, *inward*, *outward*). Os valores *bottom* e *top* alinham os textos na base ou no topo do ponto de referência do texto. Enquanto os valores *inward* e *outward* funcionam no mesmo sentido que em *hjust*, porém eles controlam o alinhamento vertical dos textos.

```
ggplot(df, aes(x, y)) +
  geom_point(color = "gray", size = 7) +
  geom_text(aes(label = text), vjust = "inward", hjust = "inward")
```



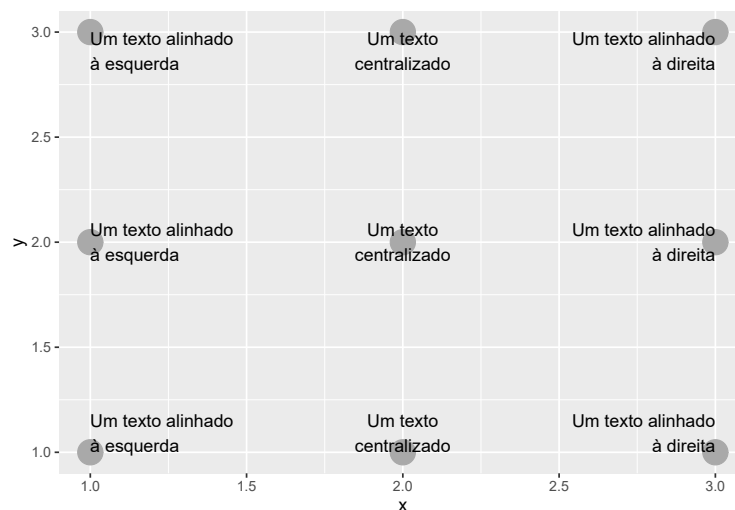
Fonte: Elaboração própria do autor.

Para deixar claro o que estes argumentos fazem, trago um novo exemplo abaixo que contém cadeias de texto de duas linhas. Caso você queira variar a justificação destes textos, ao longo do gráfico, significa que você deve conectar uma coluna de seu `data.frame` a estes argumentos em `aes()`. Porém, estes argumentos não aceitam os valores pré-definidos ao estarem dentro de `aes()`. Nes-

tas situações, você deve fornecer um número: 0 (justificado à esquerda); 0.5 (centralizado); ou 1 (justificado à direita).

```
tab <- data.frame(
  y = rep(1:3, times = 3),
  x = rep(1:3, each = 3),
  texto = rep(c("Um texto alinhado\nà esquerda",
                "Um texto\ncentralizado",
                "Um texto alinhado\nà direita"),
              each = 3
),
  hjust = rep(c(0, 0.5, 1), each = 3),
  vjust = rep(c(0, 0.5, 1), times = 3)
)

ggplot(tab, aes(x, y)) +
  geom_point(size = 7, color = "darkgray") +
  geom_text(aes(
    label = texto,
    hjust = hjust,
    vjust = vjust
  ))
```



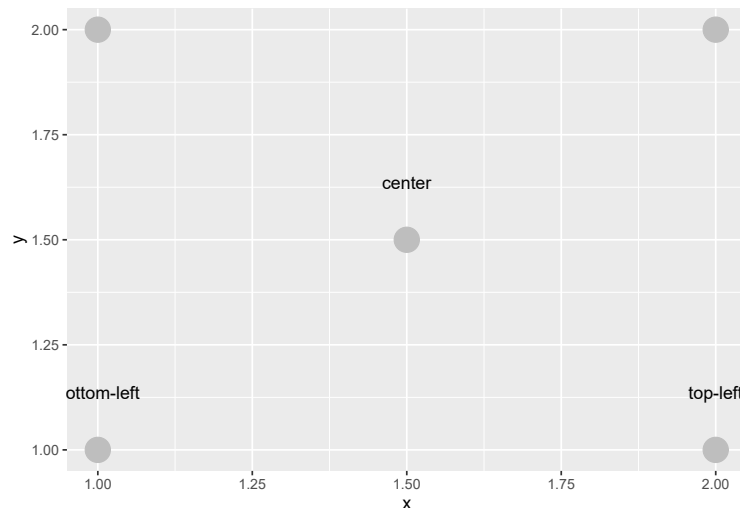
Fonte: Elaboração própria do autor.

Eu acredito que é justamente essa opção numérica, que gera toda a confusão sobre a função verdadeira destes argumentos. Pois o ggplot não gera nenhum erro caso você dê valores diferentes, e se você aumentar progressivamente estes valores, você irá perceber que o deslocamento dos textos também aumenta. Muitos que se deparam com este comportamento, podem acreditar que estes

argumentos servem para deslocar os textos, e não para alinhá-los em relação ao ponto de suas coordenadas. Por isso eu recomendo nestes argumentos, que você utilize um dos valores pré-definidos que citei anteriormente, e utilize essa escala numérica, apenas em situações em que você precisa dessa variação utilizando `aes()`.

Uma outra razão pela qual estes argumentos não são apropriados, caso você queira deslocar os textos em um sentido, é que eles não trabalham em sintonia com as escalas dos eixos. No exemplo abaixo, eu seto o valor de `vjust` para `-4`. Porém, o `ggplot` não deslocou verticalmente os textos em 4 unidades. O texto de valor `center`, por exemplo, não foi deslocado para as coordenadas ($x = 1.5$, $y = 5.5$), e se você quiser que ele chegue nessa coordenada? O que você faz? Triplica? Quadruplica o valor anterior? Tudo isso, significa que não há como você prever o quanto o texto irá deslocar, e por isso, você pode perder muito tempo testando diversos valores em um argumento inadequado para o resultado que deseja.

```
ggplot(df, aes(x, y)) +  
  geom_point(color = "gray", size = 7) +  
  geom_text(aes(label = text), vjust = -4)
```



Fonte: Elaboração própria do autor.

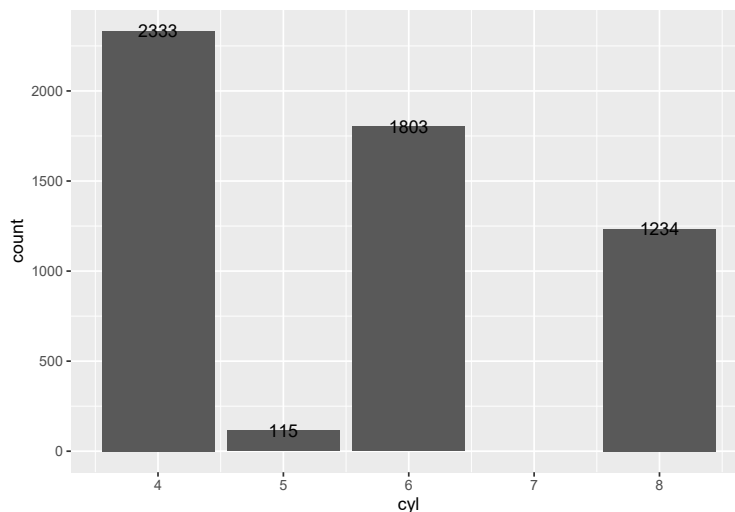
Agora, vou explicar como os argumentos de posição funcionam. Como o próprio sufixo deles dá a entender, o `nudge_y` irá deslocar verticalmente os textos, e `nudge_x`, vai deslocá-los horizontalmente. O verbo *nudge* em inglês se refere ao ato de “cutucar”, ou empurrar gentilmente alguém, logo, estes argumentos servem para “empurrar” os textos de suas posições originais no plano cartesiano. Para demonstrarmos a sua aplicação, vamos tentar rotular um gráfico de barras, que apresente um somatório da quilomentragem em cada cilindro.

Como descrevi anteriormente, o `geom_bar()` é um geom coletivo, enquanto os geom's de texto são geom's individuais. Por isso, caso você adicionar diretamente um `geom_text()` ao `geom_bar()`,

sem levar em conta essa diferença, ele irá rotular cada uma das observações da base resumidas em cada barra, e não o total que ela representa.

Para rotular corretamente essas barras, você tem duas opções: 1) calcular o somatório em um objeto separado, e em seguida fornecer este objeto ao argumento *data*, e ajustar o *aesthetic mapping* de acordo com este objeto, em `geom_text()`; ou 2) usar as transformações estatísticas que o `ggplot` já disponibiliza para esse trabalho. No exemplo abaixo, estou demonstrando a opção 1, mas darei um exemplo da opção 2 quando chegarmos à seção de transformações estatísticas do `ggplot`.

```
somatorios <- mpg %>%  
  group_by(cyl) %>%  
  summarise(soma = sum(hwy))  
  
ggplot() +  
  geom_bar(  
    mapping = aes(x = cyl, weight = hwy),  
    data = mpg  
  ) +  
  geom_text(  
    mapping = aes(x = cyl, y = soma, label = soma),  
    data = somatorios  
  )
```

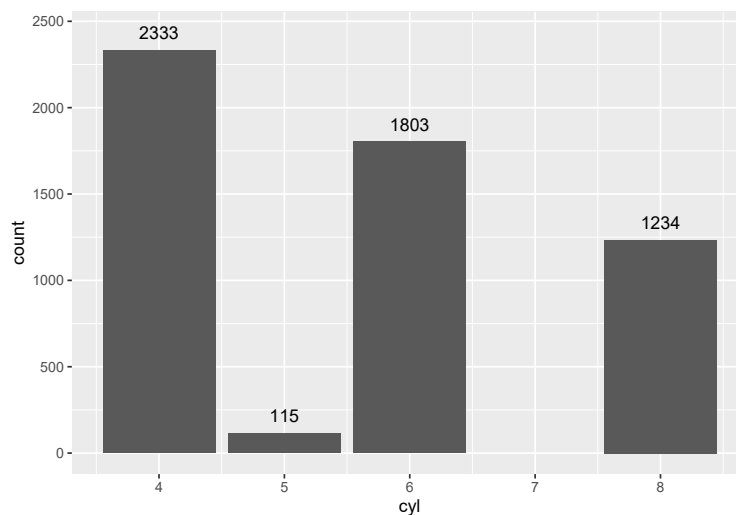


Fonte: Elaboração própria do autor.

Portanto, neste exemplo as duas camadas de `geom` utilizam não apenas *aesthetic mapping*'s, mas também fontes de dados, diferentes. Como você pode reparar acima, os rótulos estão sobre o topo da barra. Por isso, eu posso utilizar o `nudge_y` para adicionar um pequeno desvio vertical nestes rótulos, dando assim um maior espaço entre ele e a barra.

Diferentemente dos argumentos de alinhamento, os argumentos de posição (*nudge_y* e *nudge_x*) funcionam em sintonia com a escala dos eixos. Como a escala do eixo y termina em aproximadamente 2500, um desvio de 100 é provavelmente suficiente. Isso significa que caso o limite dessa escala fosse 1 décimo disso (250), por exemplo, um desvio de 100 em *nudge_y* iria gerar um deslocamento considerável nestes rótulos.

```
ggplot() +
  geom_bar(
    mapping = aes(x = cyl, weight = hwy),
    data = mpg
  ) +
  geom_text(
    mapping = aes(x = cyl, y = soma, label = soma),
    data = somatorios,
    nudge_y = 100
  )
```

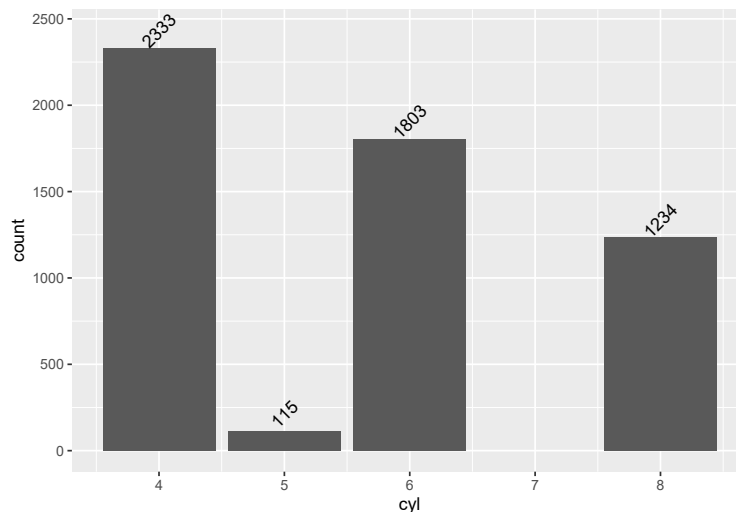


Fonte: Elaboração própria do autor.

Além dessas opções, caso você insira textos de 2 ou mais linhas no gráfico, você pode se interessar em reduzir ou aumentar o espaço entre-linhas destes textos. Neste caso, você pode controlar este espaço pelo argumento *lineheight* que define a proporção em relação à altura das letras. Um outro ponto possível de customização, é o ângulo dos textos, que é definido pelo argumento *angle*. Neste argumento, basta fornecer um número (de 0 a 360) que represente o ângulo desejado.

```
ggplot() +
  geom_bar(
    mapping = aes(x = cyl, weight = hwy),
```

```
data = mpg
) +
geom_text(
  mapping = aes(x = cyl, y = soma, label = soma),
  data = somatorios,
  nudge_y = 100,
  angle = 45
)
```



Fonte: Elaboração própria do autor.

8.7 Exportando os seus gráficos do ggplot

Após gerar os seus gráficos com o ggplot, você provavelmente vai querer exportá-los para algum arquivo de imagem. Dessa forma, você possa inserí-los em seu artigo em Word (.docx), ou no *dashboard* que você deve apresentar ao seu chefe no dia seguinte. Para realizarmos essa tarefa, precisamos utilizar funções que possam construir esses arquivos de imagem, no qual podemos guardar os nossos gráficos. Com isso, temos duas alternativas mais comuns, que são:

- 1) Uma forma mais “moderna” de exportação, através do uso da função `ggsave()`, que é exposta por Wickham (2016, Seção 8.5).
- 2) A forma tradicional de se exportar gráficos no R, descrita por Murrell (2006, Cáp. 1).

Uma outra referência que também descreve ambas alternativas, se encontra em Chang (2012, Cáp. 14). A primeira alternativa, seria uma forma mais “moderna” de exportar os seus gráficos no R, através da função `ggsave()` (uma função do próprio pacote ggplot), que tem se popularizado bastante nos últimos tempos. Entretanto, essa função nada mais é, do que um *wrapper* sobre as funções do

pacote `grDevices`, que são utilizadas na segunda alternativa apresentada acima. Ou seja, a função `ggsave()` é apenas um atalho para o método descrito por [Murrell \(2006\)](#), que utiliza as funções disponíveis no pacote `grDevices`.

Em mais detalhes, o pacote `grDevices` (que está incluso nos pacotes básicos da linguagem) oferece um conjunto de funções capazes de acessar diversos *devices* gráficos. Cada *device* gráfico, representa uma *engine* diferente que vai ser responsável por construir o arquivo onde o seu gráfico será guardado. Portanto, cada uma dessas *engines*, vão gerar um tipo arquivo diferente, ou em outras palavras, arquivos de extensões diferentes. Você já utiliza muitos desses *devices* gráficos, praticamente o tempo todo em sua rotina. Você apenas não sabia, que esse era o nome técnico dado às *engines*, que normalmente constroem esses tipos de arquivos. Sendo os exemplos mais famosos, os arquivos: JPEG/JPG (`.jpeg`, `.jpg`), PNG (`.png`), SVG (`.svg`) e PDF (`.pdf`).

8.7.1 Tipos de representação geométrica em *devices* gráficos

Ao longo das décadas passadas mais recentes, a área da computação gráfica desenvolveu diversos modelos computacionais que fossem capazes de representar visualmente e virtualmente, o nosso mundo real ([HUGHES et al., 2014](#)). Com isso, eu quero destacar que nós possuímos hoje, formas diferentes de se representar uma mesma imagem em nosso computador.

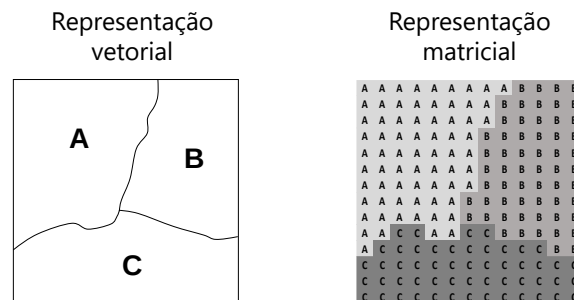
Isso significa, que cada um dos *devices* gráficos disponíveis, que podemos utilizar para guardar os nossos gráficos no R, em geral, utilizam um tipo, ou um método de representação geométrica diferente para representar a sua imagem. Cada um desses modelos, possuem características diferentes, e com isso, incorrem em diferentes erros na representação virtual de sua imagem. Logo, conhecer, mesmo que de maneira sutil, esses modelos de representação, as suas vantagens e características, se torna importante para fundamentarmos as nossas decisões sobre como vamos salvar os nossos gráficos no R.

Nós temos atualmente, dois modelos principais de representação geométrica que são utilizados para representar imagens, ao longo de toda a indústria da computação gráfica, sendo elas:

- 1) Vetorial.
- 2) Matricial.

A representação vetorial (figura 8.3), como o próprio nome dá a entender, busca conectar um conjunto de vetores (ou de linhas) para formar cada forma geométrica presente em sua imagem. Um arquivo de imagem que utiliza essa representação, possui uma descrição matemática dos elementos geométricos que compõe a sua imagem ([MURRAY; VANRYPER, 1996](#)). Essa descrição matemática possui informações como a direção, o comprimento e as coordenadas dos vetores (ou linhas) que formam cada forma geométrica de sua imagem. Em resumo, a representação vetorial funciona como aqueles desenhos infantis de “conecte os pontos”. Nesse sistema, qualquer forma presente em nosso gráfico, seja ela um quadrado, um círculo, uma letra, ou uma garrafa, é formada por um conjunto de linhas que conectam os “vértices” de cada forma.

Por outro lado, imagens que se encontram em representação matricial (figura 8.3), são popularmente

Figura 8.3: Diferenças entre as representações vetorial e matricial

Fonte: Elaboração própria do autor. Inspirado em [CÂMARA; MONTEIRO, 2001](#), p. 25.

conhecidas por *raster image*, ou *bitmap image*, e utilizam-se de uma malha quadricular (ou de um *grid*), no qual cada célula é preenchida, a fim de representar cada parte de sua imagem. Uma forma típica de identificarmos esse tipo de representação, está no efeito “escada”, ou no efeito pixelado (ou quadriculado) que adquirimos ao darmos um *zoom* nesse tipo de imagem.

Os principais *devices* gráficos disponíveis no R, que utilizam a representação vetorial, são os arquivos PDF (.pdf) e SVG (.svg). Além desses, temos alguns outros *devices* menos comuns, como os arquivos *encapsulated PostScript* (.eps) que são mais utilizados em programas da Adobe, como o PhotoShop. Imagens produzidas através de representações vetoriais, são em geral, mais bem definidas do que imagens produzidas por representações matriciais, e mesmo que o usuário dê um *zoom* grande sobre a imagem, elas são capazes de manter essa definição. Logo, como foi destacado por [Wickham \(2016, p. 185\)](#), imagens de representações vetoriais parecem mais atraentes em um número maior de lugares. Especialmente pelo fato, de que o sistema vetorial consegue representar formas geométricas (principalmente polígonos), de maneira mais precisa, do que o sistema matricial.

Apesar dessa vantagem, não são todos os programas que suportam o uso de imagens provenientes de representações vetoriais (por exemplo, o Word aceita o uso de arquivos SVG, mas não aceita o uso de PDF's para inserção de imagens). Em contrapartida, arquivos de *raster image* (ou *bitmap image*), são aceitos na grande maioria dos programas, e portanto, representam uma forma mais portátil de transportar os seus gráficos ao longo de diversos programas e sistemas. Tendo isso em mente, os *devices* gráficos mais conhecidos, que usam a representação matricial, são os arquivos PNG (.png), JPEG/JPG (.jpeg) e TIFF (.tiff).

Logo, ao escolher o *device* gráfico que irá gerar o seu arquivo de imagem, você deve refletir sobre qual o formato que mais se adequa às suas necessidades. Mesmo que você possa produzir imagens mais fiéis através de uma representação vetorial, isso não se configura na maioria das ocasiões, como uma grande vantagem. Pois, você pode se aproveitar da maior flexibilidade dos *devices* de

representação matricial, e ainda sim, produzir imagens muito bem definidas e de alta resolução. Sobre tudo com o uso de um arquivo PNG (.png) ou TIFF (.tiff), que produzem em geral, resultados melhores do que um arquivo JPEG/JPG (.jpeg).

Em resumo, caso o uso de um arquivo PDF (.pdf), ou SVG (.svg), não represente uma limitação para o seu trabalho, você geralmente vai preferí-los em detrimento de outros *devices* gráficos. Entretanto, caso você precise de uma maior portabilidade de seu gráfico, você ainda pode atingir ótimos resultados com um *device* gráfico de representação matricial, como um arquivo PNG (.png) ou TIFF (.tiff). Basta que você utilize uma resolução alta, e aplique um *anti-aliasing* sobre o arquivo em que você irá salvar o gráfico. Um bom nível de resolução para esses tipos de arquivos, se encontra na casa dos 300 dpi, sendo essa a resolução mínima requisitada pela maioria dos jornais e revistas científicas.

Concluindo, podemos utilizar diferentes tipos de representações geométricas para guardarmos informações visuais em nosso computador, com o objetivo de representarmos virtualmente uma mesma imagem. Caso queira conhecer mais a fundo essas representações, você pode consultar Frery e Periciano (2013, Cap. 2) e Câmara e Monteiro (2001) para uma introdução útil, e para uma visão mais técnica e aprofundada, você pode consultar Hughes et al. (2014, Cáp. 7 e 17) e Murray e vanRyper (1996, Cáp. 3 e 4).

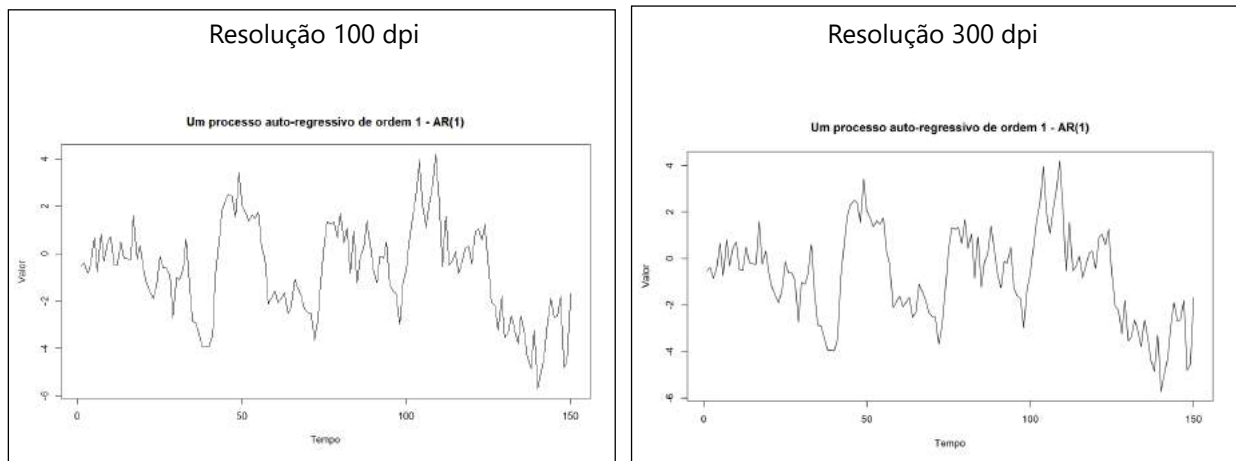
8.7.2 Pontos importantes sobre *anti-aliasing*

O *device* gráfico utilizado pelo RStudio, em seu painel de Plots, depende do sistema operacional em que você está trabalhando. No caso do Windows, o RStudio irá utilizar o *device* gráfico nativo do sistema, e infelizmente, como foi pontuado por Chase (2019), esse *device* não é muito bom. Um de seus principais problemas, é que ele não possui um mecanismo de *anti-aliasing*.

O *anti-aliasing*, é um recurso muito utilizado em diversos programas que trabalham com imagens. Um grande exemplo disso, se encontra nos *videogames*, que quase sempre possuem uma opção em suas configurações gráficas, que lhe permite aplicar esse recurso sobre o gráfico do *game*. Esse recurso, consiste em um método de suavização de imagens produzidas por representações matriciais, onde o computador tenta preencher certas áreas ao redor dos limites de cada forma geométrica representada na imagem, de forma que os contornos fiquem mais suaves e precisos, eliminando grande parte do efeito pixelado (ou quadriculado) presente em imagens desse tipo.

Portanto, se você, assim como eu, trabalha no Windows, todas as imagens (produzidas por *devices* gráficos que usam representação matricial) que você exportar no R, não vão incluir o uso de um *anti-aliasing*, por padrão do *device* gráfico utilizado pelo sistema. Isso significa, que grande parte das suas imagens, vão apresentar o efeito pixelado, mesmo que você utilize resoluções altas. Por exemplo, você pode ver na figura 8.4, especialmente na imagem com 300 dpi, que aumentar a resolução da imagem, ajuda bastante quanto ao efeito pixelado, mas que ainda não é o suficiente para eliminá-lo (se você der um *zoom* grande sobre a imagem de 300 dpi, você ainda é capaz de ver alguns resquícios do efeito “escada” que estamos tentando eliminar). Portanto, poderíamos atingir um resultado ainda melhor nessas imagens, com o uso de um *anti-aliasing*.

Figura 8.4: Aumentar a resolução de uma imagem ajuda, mas ainda não é o suficiente



Fonte: Elaboração própria do autor.

Isso é um problema particular do Windows, que ocorre sempre que utilizamos algum dos *devices* gráficos que utilizam representação matricial. Pois os *devices* gráficos utilizados pelo RStudio em outros sistemas, como Mac e Linux, apresentam “de fábrica” um mecanismo de *anti-aliasing*. Apesar desse problema, os usuários de Windows possuem uma solução simples, que é descrita por [Chase \(2019\)](#). Isto é, o uso da *engine* gráfica do Cairo Graphics, que está disponível normalmente nos sistemas Windows e oferece um recurso de anti-aliasing. O uso dessa *engine* também se torna essencial no Windows, quando desejamos utilizar em nossos gráficos, fontes que estão instaladas no nosso sistema.

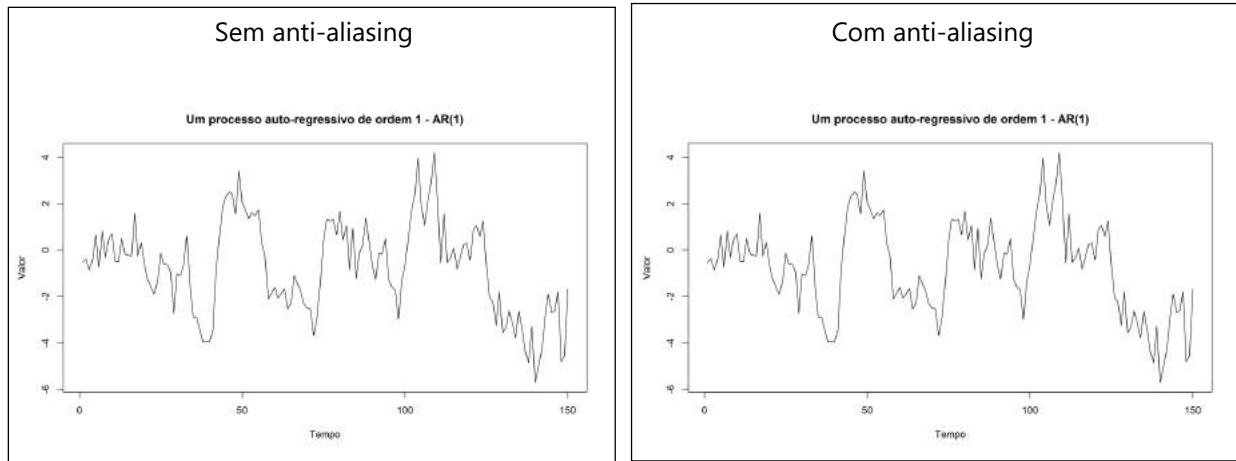
Para acessarmos a *engine* gráfica do Cairo Graphics, podemos utilizar o argumento `type`, tanto na função `ggsave()`, quanto nas funções do pacote `grDevices`. Basta igualar esse argumento ao nome `cairo`, da seguinte forma: `type = "cairo"`. No caso da função `ggsave()`, ela não possui um argumento `type` definido, mas como essa função utiliza as funções do pacote `grDevices`, o argumento `type` será repassado às funções do pacote `grDevices` durante a sua execução.

Para reforçarmos essa ideia, olhe para a figura 8.5. Ambos os gráficos foram salvos em um arquivo PNG, com as mesmas dimensões, e utilizando a mesma resolução (300 dpi). Se você der um *zoom* muito grande sobre ambas as imagens, você poderá perceber que o efeito pixelado está muito menor, na imagem em que o *anti-aliasing* foi aplicado, em relação a outra imagem que não o possui.

8.7.3 A função `ggsave()`

Como definimos anteriormente, a função `ggsave()` do pacote `ggplot`, representa apenas um atalho para o método descrito por [Murrell \(2006\)](#), sendo portanto, um método menos verboso do que o

Figura 8.5: Além de usar resoluções altas, use também anti-aliasing



Fonte: Elaboração própria do autor.

método tradicional de se exportar gráficos no R. Para utilizar a função `ggsave()`, você precisa primeiro gerar o seu gráfico, ou melhor dizendo, o seu gráfico deve estar aparecendo na área direita e inferior do seu RStudio, na seção de Plots. Pois é a partir do *cache* dessa seção, que a função irá extrair o seu gráfico, e portanto, salvá-lo em algum local de seu computador.

Dessa forma, o código necessário para o uso dessa função, vai em geral, ser semelhante ao código abaixo. Você primeiro gera o gráfico, e em seguida, utiliza a função `ggsave()`, para salvar o gráfico correspondente.

```
ggplot(mpg, aes(displ, cty)) + geom_point()  
ggsave("output.pdf")
```

O primeiro argumento (`filename`) da função `ggsave()`, corresponde ao nome que você deseja dar ao arquivo onde seu gráfico será salvo. O segundo argumento (`device`), é onde você irá selecionar o *device* gráfico desejado para o arquivo onde o gráfico será salvo. Vale ressaltar, que você não precisa definir esse argumento. Pois você pode escolher implicitamente o *device* desejado, através da extensão que você define no nome do arquivo - no primeiro argumento (`filename`). Ou seja, se no primeiro argumento, eu colocar o nome do arquivo como `output.pdf`, devido a extensão `.pdf` ao final do nome, a função `ggsave()` vai gerar um arquivo PDF para você. Mas caso o nome do arquivo seja `output.png`, a função `ggsave()` vai construir um arquivo PNG. E assim por diante. Em resumo, você pode utilizar em todos os sistemas operacionais, através da função `ggsave()`, as seguintes extensões:

- 1) `eps` - *encrypted PostScript*.
- 2) `ps` - *PostScript*.

- 3) tex - PicTex.
- 4) pdf - *Portable Document Format* (PDF).
- 5) jpeg - Arquivo JPEG.
- 6) tiff - *Tag Image File Format* (TIFF).
- 7) png - *Portable Network Graphics* (PNG).
- 8) bmp - *Bitmap Image File* (BMP).
- 9) svg - *Scalable Vector Graphics* (SVG).

Caso você deseje salvar o arquivo do gráfico, em um diretório diferente de seu diretório de trabalho atual do R, você pode utilizar o terceiro argumento (*path*), para selecionar uma pasta. Basta que você forneça um caminho absoluto até a pasta. Por exemplo, caso eu queira salvar o arquivo em minha pasta de Gráficos, localizada em minha pasta de Pesquisa, eu posso utilizar os seguintes comandos:

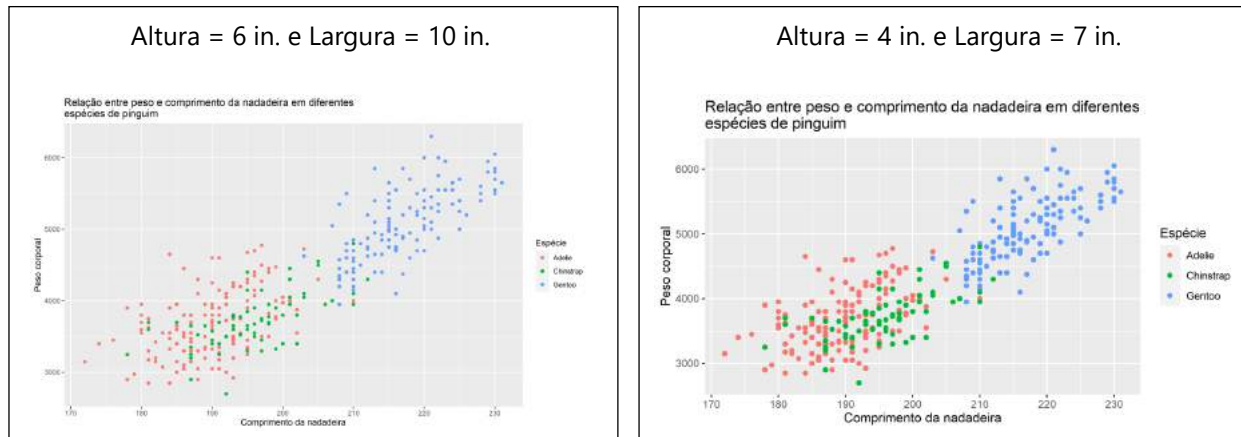
```
ggplot(mpg, aes(displ, cty)) + geom_point()
ggsave(
  filename = "output.pdf",
  path = "C:/Users/Pedro/Pesquisa/2020-08/Gráficos/"
)
```

Outros argumentos muito importantes a serem utilizados, são os argumentos *width*, *height* e *dpi*, que definem a largura, a altura e a resolução do arquivo resultante, respectivamente. É importante frisar que os argumentos *width* e *height*, trabalham com a unidade de polegadas (*inches* - *in*), sendo uma unidade menos comum em imagens. Como uma dica, você pode primeiro imaginar a largura e altura de sua imagem, em *pixels*, que é uma unidade mais comumente utilizada em situações como essa, e em seguida, converter esses *pixels* para polegadas (1 polegada equivale a 60 *pixels*), encontrando assim, o valor que você deseja fornecer aos argumentos supracitados.

Dando prosseguimento à descrição, os argumentos *width* e *height* são muito importantes, pois eles afetam diretamente a escala (ou o *aspect ratio*) da imagem. Dito de outra forma, esses argumentos acabam afetando a disposição dos elementos do gráfico, ao longo do espaço da imagem resultante. Com isso, o uso desses argumentos, envolve encontrar um certo equilíbrio, ou uma relação entre a altura e a largura da imagem, que melhor represente o seu gráfico. Por exemplo, os dois gráficos mostrados na figura 8.6, foram salvos utilizando-se a função *ggsave()*. Ambos os gráficos, foram salvos em um arquivo PNG (.png), e utilizaram uma resolução de 300 de dpi. A única diferença entre esses gráficos, se encontra nos valores de altura e largura utilizados em cada imagem.

Portanto, ao aumentarmos a altura e a largura da imagem, o gráfico resultante tende a ser mais “disperso”, e os seus elementos, menores. Essa característica é relevante, pois nós geralmente desejamos evitar um gráfico muito “disperso”, e com elementos muito pequenos. Isso se deve à função que um gráfico usualmente cumpre em uma análise. Nós frequentemente utilizamos gráficos, para nos comunicar com o nosso leitor, ao mostrarmos de forma visual, informações que são relevantes

Figura 8.6: Efeitos da relação entre altura e largura com `ggsave()`, sobre uma imagem



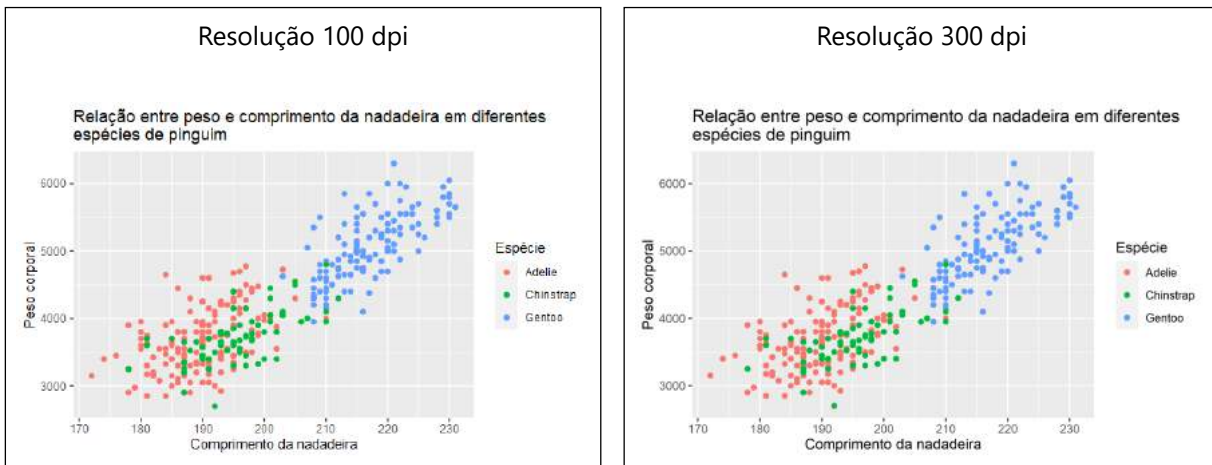
Fonte: Elaboração própria do autor.

e que trazem novas perspectivas e questões sobre uma determinada análise. Se essas informações ficam menores e muito “dispersas” ao longo do espaço do nosso gráfico, o nosso leitor tem maior dificuldade de enxergar o padrão geral (ou a informação principal) do nosso gráfico. Não apenas porque a sua visão precisa cobrir um espaço mais amplo da tela, mas também porque as formas geométricas que representam os nossos dados, podem ficar muito pequenas, e com isso, mais difíceis de se identificar.

Por outro lado, a resolução (argumento `dpi`) definida na função `ggsave()`, funciona somente com *devices* gráficos que utilizam representações matriciais (e.g. PNG, TIFF e JPEG/JPG). A resolução da imagem, é responsável por modificar apenas a dimensão da matriz, ou da malha quadricular que será utilizada para representar a sua imagem. Resoluções maiores, vão utilizar matrizes de maiores dimensões (ou em outras palavras, uma matriz com maior número de células) para representar o seu gráfico, e portanto, a imagem resultante será mais precisa, e irá sofrer de maneira menos intensa com o efeito “pixelado” produzido por representações matriciais.

Como exemplo prático, veja as imagens dos gráficos na figura 8.7. Ambos os gráficos foram salvos em um arquivo PNG, e utilizaram os mesmos valores de altura e largura. Porém, foi aplicado diferentes valores de resolução em ambas as imagens. Se você der um *zoom* sobre as imagens, você irá perceber que a imagem de 100 dpi, sofre de maneira muito mais acentuada do efeito granulado (ou pixelado), em relação a imagem de 300 dpi.

Lembre-se que podemos melhorar ainda mais a aparência dessas imagens, ao utilizarmos um recurso de *anti-aliasing*. Como já definimos na seção de **Pontos importantes sobre anti-aliasing**, para utilizarmos tal recurso, precisamos acessar a *engine* do Cairo Graphics, e para isso, precisamos

Figura 8.7: Efeitos da resolução com ggsave() sobre uma imagem

Fonte: Elaboração própria do autor.

apenas definir o argumento `type` para o nome `cairo`.

```
ggplot(mpg, aes(displ, cty)) + geom_point()
ggsave(
  filename = "output.png",
  path = "C:/Users/Pedro/Pesquisa/2020-08/Gráficos/",
  width = 7,
  height = 5,
  type = "cairo"
)
```

8.7.4 A forma tradicional de se exportar gráficos no R

Apesar da função `ggsave()` ser um atalho útil, eu (Pedro) particularmente prefiro usar diretamente as funções do pacote `grDevices`, sempre que desejo exportar algum gráfico produzido no R. Parte dessa preferência, reside no fato de que a função `ggsave()` não oferece até o momento, suporte para a função `cairo_pdf()`, que se torna essencial quando desejamos exportar gráficos que utilizam fontes personalizadas, ou que estão instaladas em nosso sistema. Vale lembrar, que o pacote `grDevices` já está incluso nos pacotes básicos do R, e por essa razão, ele é carregado automaticamente em toda sessão do R que você inicia.

Como é descrito por Murrell (2006, Seção 1.3), o processo tradicional de exportação de gráficos no R, é bem simples, e envolve três passos diferentes: 1) abrir um arquivo construído por algum *device* gráfico; 2) gerar o seu gráfico; 3) fechar o arquivo produzido pelo *device* gráfico.

Portanto, no primeiro passo, vamos criar um novo arquivo de imagem (vazio) em nosso computador, de acordo com um *device* gráfico de nossa preferência. Dessa forma, o arquivo fica em aberto, a espera de algum *input* gráfico a ser armazenado. Em seguida, nós podemos gerar o nosso gráfico. Sendo que diferentemente da função `ggsave()`, quando abrimos um arquivo de imagem (como fizemos no passo 1), qualquer gráfico que geramos não será mostrado no painel direito e inferior (seção Plots) do nosso RStudio. Pois ele é diretamente levado para o arquivo que abrimos. Por último, podemos fechar o arquivo que abrimos no primeiro passo, encerrando dessa forma, o processo de exportação.

Para abrimos um novo arquivo de imagem em nosso computador, temos as funções disponíveis abaixo. Perceba que a lista de arquivos abaixo, é praticamente idêntica à lista que mostramos na seção anterior. Pois como já destacamos anteriormente, a função `ggsave()` vai utilizar “por trás dos bastidores”, todas essas funções abaixo (exceto a função `svg()`¹³) para construir os seus arquivos de imagem.

- 1) `postscript()` - *encrypted PostScript e PostScript*.
- 2) `pictex()` - *PicTex*.
- 3) `pdf()` e `cairo_pdf()` - *Portable Document Format (PDF)*.
- 4) `jpeg()` - *Arquivo JPEG*.
- 5) `tiff()` - *Tag Image File Format (TIFF)*.
- 6) `png()` - *Portable Network Graphics (PNG)*.
- 7) `bmp()` - *Bitmap Image File (BMP)*.
- 8) `svg()` - *Scalable Vector Graphics (SVG)*.

Independente de qual o *device* gráfico, ou a função que você escolher para abrir um arquivo em seu computador, você irá fechar esse arquivo (terceiro passo), por meio da função `dev.off()`. Dessa forma, o código necessário para gerarmos, por exemplo, um arquivo PNG, através desse método de exportação, é semelhante aos comandos abaixo. De certa forma, você utiliza as funções do pacote `grDevices`, de modo que elas “contornem”, ou “envolvam” os comandos que geram o seu gráfico.

```
# Abra um arquivo de imagem
# com algum device gráfico
png("output.png")

# Construa algum gráfico
ggplot(mpg, aes(displ, cty)) + geom_point()

# Feche o arquivo que você criou
# com dev.off()
dev.off()
```

¹³No caso de arquivos do tipo SVG, a função `ggsave()` utiliza a função `svglite()`, que vem do pacote `svglite`.

Assim como na função `ggsave()`, o primeiro argumento (`filename` ou `file`) de todas as funções do pacote `grDevices`, é responsável por definir o nome do arquivo onde o seu gráfico será salvo. Porém, as semelhanças com a função `ggsave()` acabam por aqui.

Diferentemente da função `ggsave()`, você precisa ficar mais atento ao definir as dimensões de sua imagem nas funções do pacote `grDevices`, pois as unidades utilizadas nos argumentos `height` e `width` ao longo dessas funções, variam. Uma boa forma de guardar essas unidades, é categorizar as funções de acordo com a representação geométrica que elas utilizam. As funções que utilizam representações vetoriais (PDF, SVG e EPS) usam a unidade de polegadas (*inches*), para definir as dimensões de sua imagem. Já as funções que utilizam representações matriciais (PNG, JPEG/JPG, TIFF, BMP) usam a unidade de *pixels*.

Uma outra diferença presente nas funções do pacote `grDevices`, é que o argumento responsável por definir a resolução da imagem, se chama `res` (abreviação para *resolution*), ao invés de `dpi` como ocorre em `ggsave()`. Entretanto, a unidade utilizada no argumento `res`, permanece a mesma, em relação ao argumento `dpi`.

8.7.4.1 Arquivos PNG, JPEG/JPG, TIFF e BMP

Para os exemplos dessa seção, vou utilizar a função `png()`, com o objetivo de criar um modelo guia, sobre como você pode configurar esse conjunto de funções, que se referem a *devices* gráficos que utilizam representações matriciais. Ou seja, você pode replicar normalmente o método de uso da função `png()`, ou os seus argumentos, para as demais funções desse conjunto (que funcionam de maneira idêntica), basta trocar a função `png()` por uma dessas funções: `jpeg()`, `tiff()` e `bmp()`.

Em todas as ocasiões que você utilizar uma dessas funções, você possui ao menos 5 argumentos que você provavelmente irá definir. O primeiro argumento (`file` ou `filename`) de todas essas funções, é onde você irá definir o nome do arquivo, em que você está salvando o seu gráfico. Além dele, temos também os dois argumentos que definem a largura (`width`) e a altura (`height`) do arquivo resultante (lembre-se que esses argumentos trabalham com a unidade de *pixels*). Em seguida, temos o argumento `res`, que é responsável por definir a resolução do arquivo de imagem. Por último, mas não menos importante, temos o argumento `type`, que é responsável por definir se o R irá utilizar o *device* gráfico nativo do sistema, ou a *engine* do Cairo Graphics para construir a sua imagem.

Tendo esses argumentos em mente, temos logo abaixo um código modelo, sobre como poderíamos configurar essa função. No caso de arquivos PNG, você ainda pode utilizar o valor `cairo-png` no argumento `type`, para utilizar uma outra *engine* interna do Cairo Graphics. Porém, os resultados produzidos por `cairo` e `cairo-png` através do argumento `type`, são virtualmente idênticos (ao menos a olho nu). Você pode encontrar mais detalhes sobre a diferença entre esses dois métodos, na documentação da função `png()` (execute o comando `?png` no console para acessar essa documentação).

```
png(  
  filename = "um_nome_qualquer.png",  
  width = 2800,
```

```
height = 1800,  
res = 300,  
type = "cairo"  
)  
ggplot(mpg, aes(displ, cty)) + geom_point()  
dev.off()
```

Agora, é muito importante destacar que a resolução da imagem (que você define no argumento `res`) construída por essas funções, possui um efeito diferente do que vimos na função `ggsave()`. Quando estávamos discutindo a função `ggsave()`, vimos que a resolução definida no argumento `dpi` dessa função, cumpria o trabalho para o qual foi desenvolvida, que consistia no aumento ou redução da malha quadricular responsável pela representação da imagem. Em outras palavras, a resolução afetava diretamente a precisão da imagem construída.

Porém, em comparação com a função `ggsave()`, a resolução definida no argumento `res`, possui um efeito extra nas funções `png()`, `jpeg()`, `tiff()` e `bmp()`. Em resumo, ao modificarmos a resolução nessas funções, nós também afetamos o espaço da imagem, da mesma forma como ocorre ao modificarmos a altura e a largura da imagem. Por isso, ao utilizar essas funções, você possui mais um item a considerar, ao procurar pelo equilíbrio que melhor representa o seu gráfico.

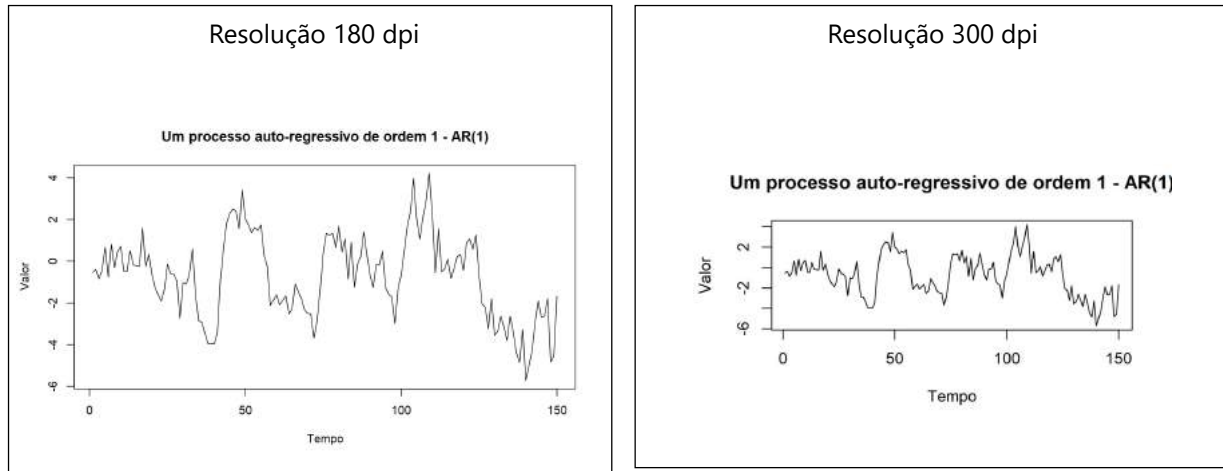
Descrevendo esse efeito em mais detalhes, nas funções `png()`, `jpeg()`, `tiff()` e `bmp()`, caso nós mantivermos a altura e a largura da imagem constantes, ao aumentarmos a sua resolução, estamos de certa forma comprimindo o gráfico a um espaço menor. Por outro lado, ao reduzirmos essa resolução, estamos produzindo o efeito contrário, e como resultado, nós aumentamos o espaço que o gráfico irá ocupar na imagem.

Isto significa, que para você utilizar altos níveis de resolução em suas imagens, você terá que compensar os efeitos dessa resolução, com maiores valores para a altura e largura de sua imagem. Veja por exemplo, as imagens mostradas na figura 8.8. Ambas imagens apresentam exatamente o mesmo gráfico. Sendo que ambos os gráficos foram salvos em um arquivo PNG (construídos pela função `png()`), e utilizaram as mesmas dimensões (altura = 900 *pixels*, largura = 1500 *pixels*). Entretanto, os dois arquivos de imagem usaram resoluções diferentes. Perceba que o gráfico presente na imagem com maior resolução (300 dpi), está mais “comprimido”, enquanto o gráfico com menor resolução (180 dpi) traz um visual mais “natural”, como se o gráfico tivesse um espaço mais ideal para ocupar na imagem.

8.7.4.2 Arquivos PDF e SVG

Nessa seção, vamos discutir três funções utilizadas para construirmos dois tipos de arquivos de imagem, que utilizam representações vetoriais, mais especificamente PDF (`pdf()` e `cairo_pdf()`) e SVG (`svg()`).

Você pode configurar as funções `pdf()`, `cairo_pdf()` e `svg()`, de maneira muito parecida com a função `ggsave()`. Dessa forma, você possui três argumentos principais a serem tratados nessas funções. O argumento `file`, para dar um nome ao arquivo que você está criando. E os argumentos

Figura 8.8: Efeitos da resolução em `png()` sobre uma imagem

Fonte: Elaboração própria do autor.

`height` e `width` para definir a altura e a largura da imagem. Lembre-se que para as funções de *drives* gráficos que utilizam representação vetorial, as dimensões da imagem são definidas em polegadas (*inches*), e não em *pixels*. Abaixo temos um exemplo de uso dessas funções.

```
pdf("output.pdf", width = 8, height = 5)
ggplot(mpg) + geom_point(aes(x = displ, y = hwy))
dev.off()

cairo_pdf("output.pdf", width = 8, height = 5)
ggplot(mpg) + geom_point(aes(x = displ, y = hwy))
dev.off()

svg("output.svg", width = 8, height = 5)
ggplot(mpg) + geom_point(aes(x = displ, y = hwy))
dev.off()
```

Curiosamente, se você está exportando um arquivo PDF, você pode salvar múltiplos gráficos em um mesmo arquivo, de modo que cada gráfico terá a sua própria página. Como [Chang \(2012, p. 324\)](#) descreve, caso você abra um arquivo PDF (com a função `pdf()` ou `cairo_pdf()`), e gere mais de um gráfico antes de encerrar esse arquivo com a função `dev.off()`, cada gráfico gerado terá a sua própria página no arquivo resultante.

Para fazer isso, nenhuma nova configuração é necessária sobre a função `pdf()`. Logo, independentemente de quantos gráficos você esteja planejando guardar nesse arquivo, você não precisa alterar nenhum argumento, em relação aos comandos anteriores. Tudo o que você precisa fazer, é abrir um

novo arquivo com a função, e criar quantos gráficos você desejar antes de fechar o arquivo.

```
pdf("output.pdf", width = 8, height = 5)

# Gráfico 1
ggplot(mpg) + geom_point(aes(x = displ, y = hwy))
# Gráfico 2
ggplot(mpg) + geom_bar(aes(x = cyl))
# Gráfico 3
ggplot(mpg) + geom_histogram(aes(x = hwy), color = "black")

dev.off()
```

Porém, para atingir esse mesmo resultado com a função `cairo_pdf()`, você precisa ainda adicionar um novo argumento, chamado `onefile`. Tudo o que você deve fazer, é configurar esse argumento para verdadeiro (`TRUE`), como no exemplo abaixo. Dessa forma, todos os gráficos gerados por você, serão guardados em um mesmo arquivo.

```
cairo_pdf("output.pdf", width = 8, height = 5, onefile = TRUE)

# Gráfico 1
ggplot(mpg) + geom_point(aes(x = displ, y = hwy))
# Gráfico 2
ggplot(mpg) + geom_bar(aes(x = cyl))
# Gráfico 3
ggplot(mpg) + geom_histogram(aes(x = hwy), color = "black")

dev.off()
```

Como o próprio nome da função `cairo_pdf()` dá a entender, essa função utiliza a *engine* gráfica do Cairo Graphics para construir o seu arquivo PDF. Como o recurso de *anti-aliasing* só é aplicado sobre imagens produzidas por representações matriciais, o uso do Cairo Graphics possui um papel diferente em representações vetoriais. Em resumo, você só vai precisar da função `cairo_pdf()`, caso você tenha utilizado em seu gráfico, fontes personalizadas, ou que estão instaladas em seu sistema.

Ao executar a função `pdfFonts()`, você pode encontrar uma lista, contendo as informações sobre todas as fontes atualmente disponíveis em sua sessão, que podem ser utilizadas ao exportarmos o nosso gráfico através da função `pdf()`. Em outras palavras, essa lista mostra todas as fontes que você pode utilizar em seu gráfico no R, e que vão estar disponíveis ao salvar esse gráfico em um arquivo PDF. Adicionar novas fontes a essa lista mostrada pela função `pdfFonts()`, não é algo simples. Por isso, diversos pacotes tem sido desenvolvidos com o objetivo de facilitar o uso de fontes personalizadas em gráficos no R. Dentre eles, o pacote `extrafont` é o que mais tem se destacado.

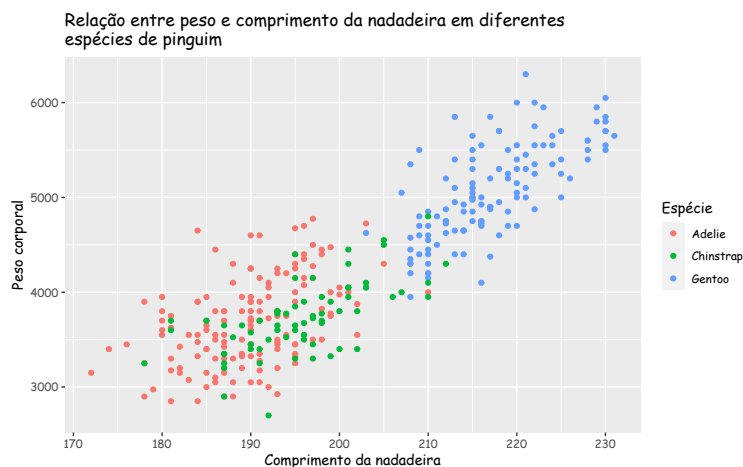
No próximo capítulo, vou mostrar como podemos aplicar esse pacote. Mas resumidamente, com o uso do pacote `extrafont`, o processo para o uso de fontes do sistema em seus gráficos no R, consiste em duas etapas: 1) “importar” as fontes para o R, de forma que ele guarde um registro da localização, e de outras informações sobre cada fonte instalada em seu sistema; 2) após o passo 1, você precisa sempre “carregar” essas fontes, durante toda sessão do R, em que você desejar utilizá-las.

Porém, no caso específico de uma exportação para um arquivo PDF, mesmo após utilizarmos as funções do pacote `extrafont`, para carregarmos as fontes que desejamos utilizar em nossa sessão do R - como demonstrado por Qiu (2015, p. 101), a função `pdf()`, ainda sim, costuma não ser capaz de incorporar as fontes utilizadas, ao arquivo PDF resultante. Apesar da dificuldade inerente ao processo, a *engine* interna do Cairo Graphics, oferece suporte para a criação de arquivos PDF, e ainda mais importante, oferece um excelente suporte para o processo de incorporação de fontes (ou *font embedding*, como é comumente chamado) de seu computador para o arquivo PDF criado.

O processo de *font embedding*, consiste em incluir dentro de seu arquivo PDF, uma descrição completa das fontes utilizadas ao longo de seu PDF. Com essa descrição, as fontes utilizadas em seu arquivo PDF, se tornam independentes do sistema no qual elas estão sendo mostradas ou impressas. Em outras palavras, as fontes de seu PDF vão ser corretamente mostradas na tela de qualquer computador, independentemente se esse computador possui ou não aquelas fontes presentes em seu sistema. O ponto forte da função `cairo_pdf()`, é que ela realiza esse processo de *font embedding* automaticamente.

Por exemplo, o gráfico mostrado na figura 8.9, utiliza a fonte Comic Sans MS (uma fonte normalmente disponível em todo sistema Windows) e foi salvo utilizando a função `cairo_pdf()`.

Figura 8.9: Um gráfico que utiliza a fonte Comic Sans MS



Fonte: Elaboração própria do autor.

Exercícios

Exercício 1

Descubra qual o problema dos comandos abaixo, porque eles não geram um gráfico conforme esperado? Ou porque eles sequer geram algum gráfico? Vale destacar que, as tabelas `mpg` e `diamonds` estão disponíveis através do próprio pacote `ggplot2`. Portanto, assim que você chamar por esse pacote em sua sessão através do comando `library()`, você terá acesso a essas tabelas.

- A) Os comandos abaixo deveriam gerar um simples gráfico de dispersão, porém, um erro é criado. Porque esse erro ocorre? Copie e cole em seu R e veja esse erro com seus próprios olhos.

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

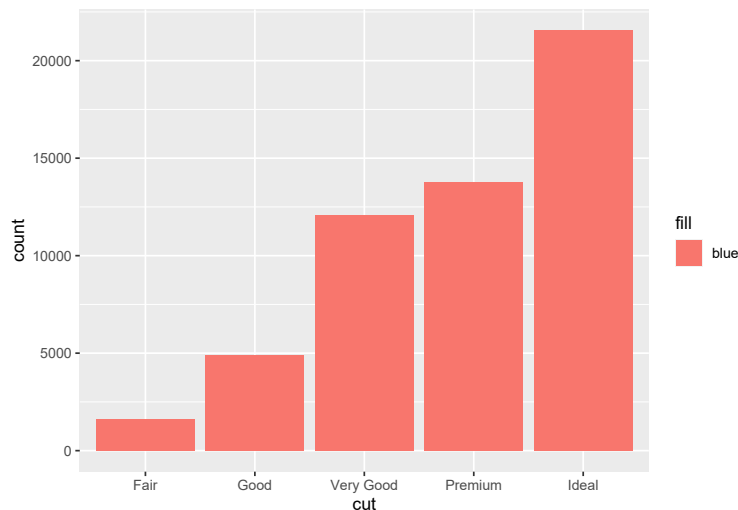
- B) Os comandos abaixo deveriam gerar um gráfico de dispersão, onde os pontos seriam coloridos de acordo com os valores da coluna `cut`. Porém, o resultado é um gráfico de dispersão onde todos os pontos continuam pretos! O que ocorreu de errado nesses comandos?

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

- C) Os comandos abaixo deveriam gerar um simples gráfico de barras, onde todas as barras deveriam ser coloridas pela cor azul (*blue*), porém, o resultado é um gráfico com barras coloridas de um vermelho salmão. Porque isso ocorre? Como podemos corrigir esses comandos para que todas as barras estejam coloridas de azul?¹⁴.

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

¹⁴Esse item foi inspirado em um exercício de Wickham e Grolemund (2017, p. 12).



Fonte: Elaboração própria do autor.

Exercício 2

Como exercício prático, utilize as funções do pacote ggplot para desenhar os objetos abaixo:

- A) A bandeira do movimento LGBTQ+. Como uma ajuda, nós temos abaixo um vetor contendo os códigos de cada cor presente nessa bandeira:

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

- B) Considerando a função quadrática $y = x^2 + 15x + 32$, desenhe a curva da função para o intervalo de $0 < x < 1000$.
- C) Desenhe um conjunto de setas apontando para o texto "Uma anotação muito importante". Ou seja, desenhe o texto guardado no objeto anotacao abaixo em seu ggplot e, em seguida, tente desenhar um conjunto de setas apontando para essa anotação.

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

Exercício 3

Na média qual a qualidade de corte (cut) na tabela diamonds que gera o maior preço (price). Dito de outra forma, utilize um gráfico do ggplot para responder à seguinte pergunta: tendo a

tabela diamonds em mãos, quais são os cortes descritos na coluna cut que geram os diamantes mais caros do mercado, isto é, que possuem os maiores preços na coluna price. Lembre-se que a tabela diamonds advém do próprio pacote ggplot2, logo, se você chamou por esse pacote em sua sessão com um comando library(), você já tem acesso à tabela diamonds.

```
library(ggplot2)
```

```
### Ao chamar pelo pacote
```

```
### ggplot2, você terá acesso
```

```
### à tabela diamonds
```

```
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
```

Capítulo 9

Configurando componentes estéticos do gráfico no ggplot2

9.1 Introdução e pré-requisitos

No primeiro capítulo sobre o ggplot, vimos quatro das várias camadas que compõe um gráfico estatístico segundo a abordagem de [Wilkinson \(2005\)](#). Mais especificamente, vimos as três camadas essenciais que estão presentes em qualquer gráfico: os dados utilizados (*data*), o mapeamento (*aesthetic mapping*) de variáveis de sua tabela para atributos estéticos do gráfico, e as formas geométricas (*geom's*) que representam os seus dados no gráfico. Além dessas camadas essenciais, também explicamos como você pode criar diferentes facetas de um mesmo gráfico.

Neste capítulo, estaremos focando nas outras camadas, mais especificamente, aquelas que controlam aspectos visuais e estéticos do gráfico. Estaremos utilizando novamente neste capítulo, o mesmo gráfico (`plot_exemplo`) como base para os nossos exemplos.

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

9.2 Tema (*theme*) do gráfico

O tema do gráfico, diz respeito a todos os elementos e configurações estéticos que não afetam, ou que não estão conectadas aos dados dispostos no gráfico. Ou seja, os temas não alteram as propriedades perceptíveis do gráfico, mas ajuda você a torná-lo esteticamente agradável ([WICKHAM, 2016](#), p. 169). Em outras palavras, o tema lhe dá controle sobre as fontes utilizadas, o alinhamento do texto, a grossura do *grid* e das marcações, a cor do plano de fundo do gráfico, etc.

Todos os aspectos temáticos do gráfico são configurados pela função `theme()`, que possui vários argumentos. Cada argumento dessa função, lhe permite configurar um elemento de seu gráfico. Onde cada um destes elementos, são associados a um tipo de elemento diferente. Por exemplo, o

título do gráfico, é um texto, logo, ele é associado ao elemento do tipo texto - `element_text()`, já as retas dos eixos são associadas ao elemento do tipo linha - `element_line()`.

Os tipos de elemento são apenas uma convenção, para que você saiba qual função `element_*`() é a apropriada para configurar o elemento desejado. Por exemplo, se o título do gráfico, é um elemento associado ao tipo “texto”, você deve usar a função `element_text()` para modificar este elemento. Porém, se você quer configurar o *background* do gráfico, você deve utilizar a função `element_rect()`, pois este elemento está associado ao tipo “retângulo”. Os diversos tipos de elemento são:

- 1) Texto: `element_text()`.
- 2) Retângulo: `element_rect()`.
- 3) Linha: `element_line()`.
- 4) Branco ou vazio: `element_blank()`.

Você provavelmente está se perguntando o porquê da existência de um tipo de elemento “vazio”. O jornalista americano William Chase, apresentou um ditado na última conferência internacional do RStudio, que representa bem o papel que este tipo de elemento tem a cumprir no `ggplot`. O ditado diz o seguinte:

“O espaço em branco no gráfico é como o alho que tempera a sua comida. Pegue o tanto que você acha necessário, e então triplique essa quantidade”. William Chase, *The Glamour of Graphics*, `rstudio::conf`, 2020.

A noção de espaço, é muito importante no seu gráfico, seja porque você tem itens que estão tomando muito espaço das formas geométricas que estão representando os seus dados no gráfico, ou então, porque você quer tornar a visão de seu gráfico mais leve (ou mais “dispersa”) para o leitor. Por isso, o elemento do tipo “vazio” serve para eliminar elementos que são desnecessários em seu gráfico, dando assim, maior espaço para aqueles elementos que são de fato importantes.

Ao longo dessa seção, estarei utilizando um mesmo gráfico, para exemplificar algumas das principais configurações possíveis em `theme()`. Para não repetir o código que gera o gráfico, toda vez que alterarmos algo nele, eu vou guardar este gráfico em um objeto que dou o nome de `plot_exemplo`. Dessa forma, toda vez que quiser alterar algum elemento do gráfico, basta que eu adicione a função `theme()` a este objeto, onde o gráfico está guardado.

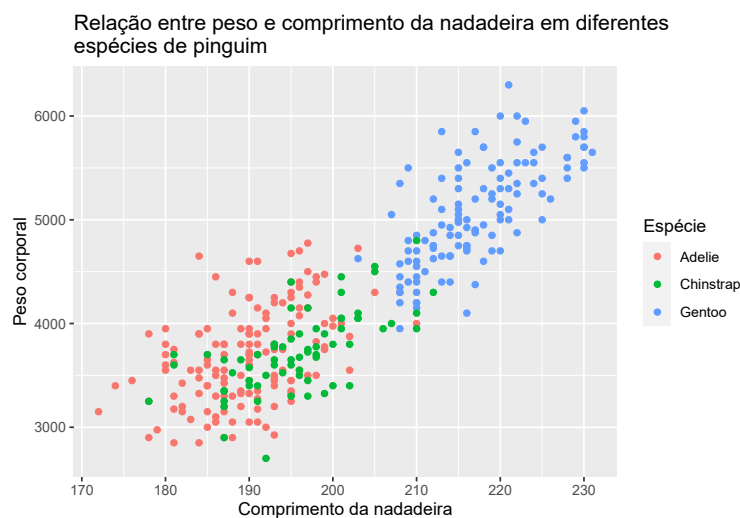
```
plot_exemplo <- ggplot(data = penguins) +  
  geom_point(  
    aes(  
      x = flipper_length_mm,  
      y = body_mass_g,  
      color = species  
    )  
  ) +  
  labs(
```

```

title = "Relação entre peso e comprimento da nadadeira
em diferentes\nespécies de pinguim",
x = "Comprimento da nadadeira",
y = "Peso corporal",
color = "Espécie"
)

print(plot_exemplo)

```



Fonte: Elaboração própria do autor.

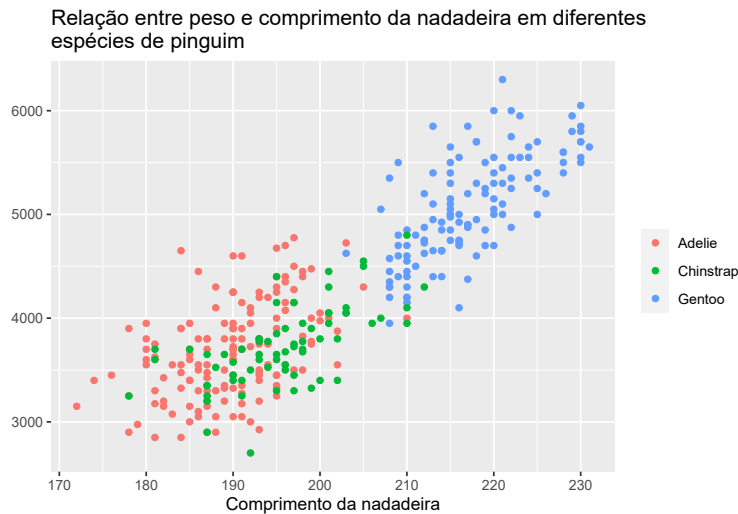
9.3 Eliminando elementos do gráfico

Como eu disse, você muitas vezes vai querer eliminar elementos desnecessários e que estão tomando muito espaço de seu gráfico. Para esta tarefa, basta utilizar `element_blank()` sobre o argumento de `theme()` que controla este elemento em questão. No exemplo abaixo, estou eliminando o título da legenda, que é controlada por `legend.title`, e também estou eliminando o título do eixo y com `axis.title.y`.

```

plot_exemplo +
  theme(
    legend.title = element_blank(),
    axis.title.y = element_blank()
  )

```



Fonte: Elaboração própria do autor.

9.4 Alterando a temática de textos

Você possui diversos elementos textuais em seu gráfico, logo abaixo, na figura 9.1, estou relacionando cada elemento textual ao seu respectivo argumento em `theme()`. Vale ressaltar, que há outros elementos textuais, como o subtítulo do gráfico, que não está presente em nosso `plot_exemplo`. Portanto, até os próprios valores do eixo são tratados como textos do gráfico. Como mencionei antes, você precisa da função `element_text()` para configurar este tipo de elemento.

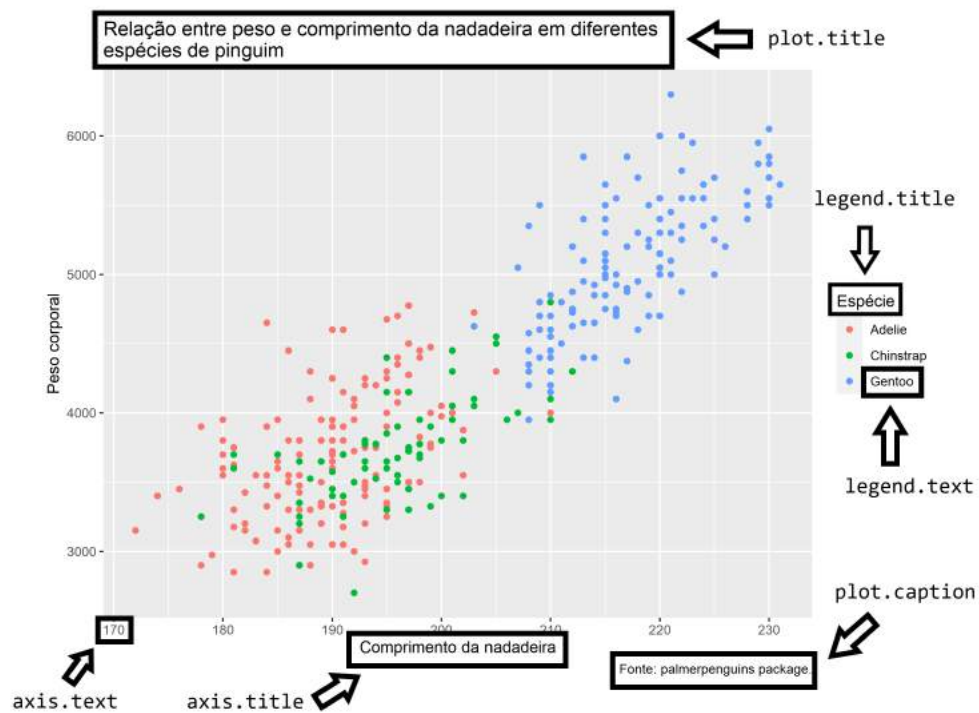
Vamos pensar primeiro no título, que é uma parte importante de seu gráfico e que deve possuir algum tipo de destaque. Por enquanto, o único fator que destaca o título do gráfico dos outros elementos textuais, é o tamanho da fonte usada. Porém, e se quiséssemos adicionar outros fatores de destaque? Como por exemplo, utilizar uma fonte em itálico, ou em negrito.

O argumento de `theme()` responsável por controlar o título do gráfico, é o `plot.title`, e portanto, utilizo a função `element_text()` sobre este argumento, para acrescentarmos novos destaques a este título. O argumento de `element_text()` que afeta o estilo da fonte (negrito, itálico, etc.) é o `face`. No exemplo abaixo, eu dou o valor `"italic"` indicando a função que use o estilo itálico sobre o título:

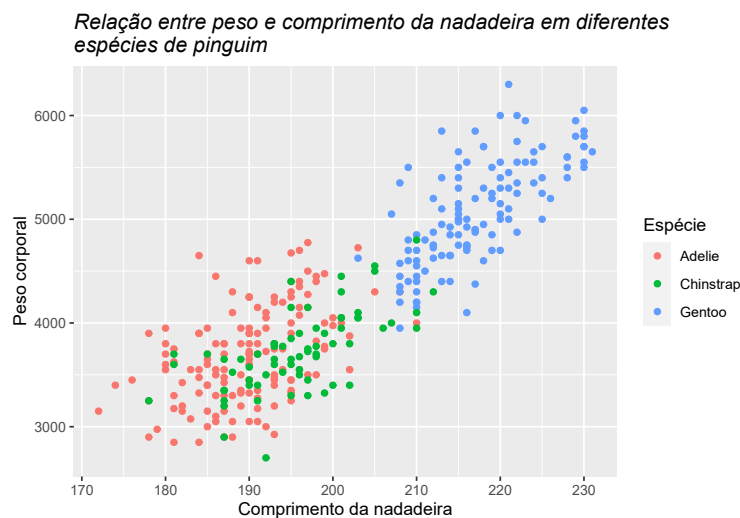
```
plot_exemplo +  
  theme(  
    plot.title = element_text(face = "italic")  
  )
```

Eu posso também destacar outras áreas do gráfico, como o título da legenda, que é controlado pelo argumento `legend.title`. Eu costumo reduzir o tamanho deste título, e colocá-lo em negrito, e para isso, utilizo os argumentos `size` e `face`. Para colocar algum texto em negrito, você deve utilizar

Figura 9.1: Principais elementos textuais do gráfico e seus respectivos argumentos na função `theme()`



Fonte: Elaboração própria do autor.

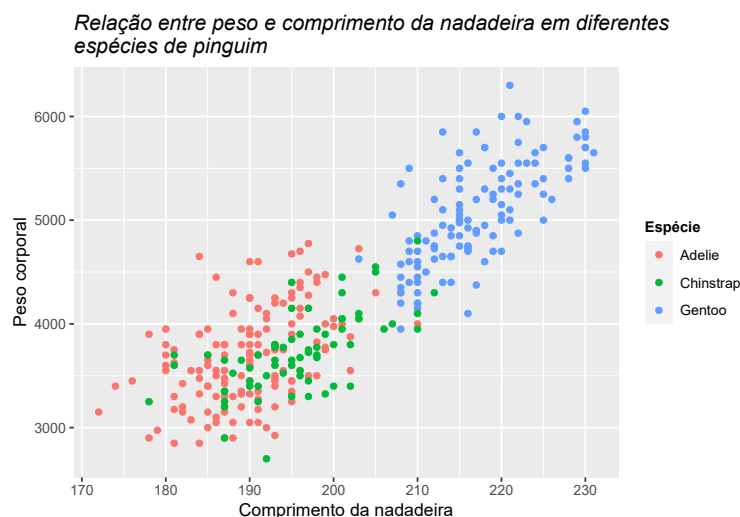


Fonte: Elaboração própria do autor.

o valor "bold", em face. Eu poderia inclusive, colocar este texto em itálico e negrito (para isso, você deve utilizar o valor "bold.italic").

Vale também destacar, que o argumento `size`, trabalha por padrão com a unidade milímetros (mm). Porém, como é um pouco contraintuitivo trabalhar com tamanho de fontes nesta unidade, eu costumo transformá-la para pontos (pt). Para isso, o ggplot oferece uma variável (`.pt`) que já contém o valor necessário para essa transformação. Assim, o que você precisa fazer é colocar o valor em pontos (pt) desejado, e dividi-lo por essa variável (`.pt`), que contém o valor necessário para a conversão. No exemplo abaixo, estou reduzindo o título da legenda ao tamanho 26 pt.

```
plot_exemplo +
  theme(
    plot.title = element_text(face = "italic"),
    legend.title = element_text(face = "bold", size = 26/.pt)
  )
```



Fonte: Elaboração própria do autor.

Além destas modificações, você talvez queira mudar o alinhamento do título do gráfico. Atualmente, você pode reparar que este título está alinhado à esquerda do gráfico, ou em outras palavras, está alinhado em relação a borda esquerda do gráfico.

Neste caso, estou me referindo ao alinhamento horizontal do título, e por isso, utilizo o argumento `hjust`. Este argumento funciona da mesma forma em que o vimos anteriormente, ele pega um número de 0 a 1. Sendo que o valor 0 representa o alinhamento totalmente à esquerda, o valor 0.5 centraliza o texto, e o valor 1 representa o alinhamento totalmente à direita. No exemplo abaixo, estou centralizando o título do gráfico.

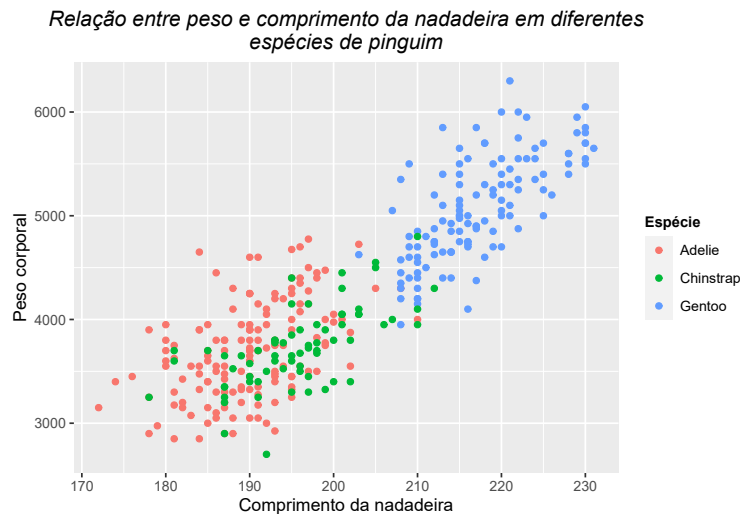
```
plot_exemplo +
  theme(
```



```

plot.title = element_text(face = "italic", hjust = 0.5),
legend.title = element_text(face = "bold", size = 26/.pt)
)

```



Fonte: Elaboração própria do autor.

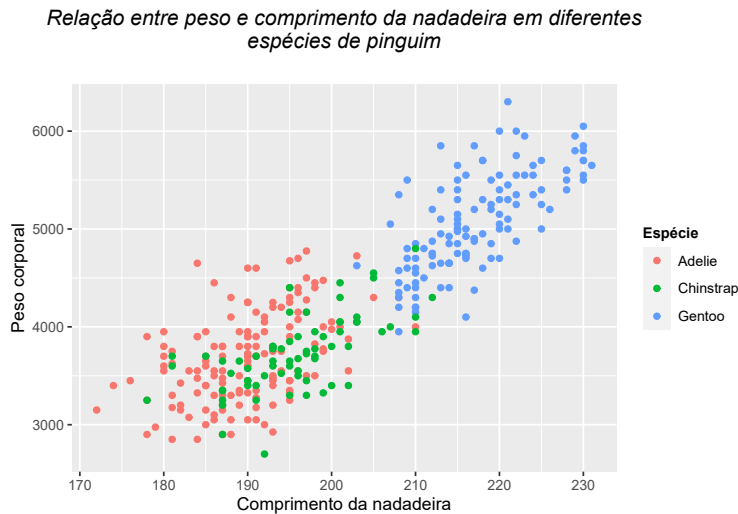
Um outro ponto, que talvez seja de seu interesse, é alterar o espaço entre os elementos do gráfico. Você pode controlar este fator, através da função `margin()`, sobre o argumento `margin` de `element_text()`. Dentro da função `margin()`, temos 4 argumentos que se referem as bordas do seu texto. Dito de outra forma, esses argumentos definem a borda do texto, na qual você deseja acrescentar o espaço: `t` (*top*) se refere ao topo do texto; `r` (*right*) se refere à direita do texto; `l` (*left*) se refere à esquerda do texto; e `b` (*bottom*) se refere à base (ou a borda inferior) do texto.

Por exemplo, podemos dar mais destaque ao título do gráfico, ao adicionar um pouco mais de espaço entre ele e a borda do gráfico. Neste caso, o gráfico está abaixo do título, logo, estamos querendo adicionar espaço na borda inferior (argumento `b`) do título do gráfico. Em seguida, basta que eu defina no argumento, quanto de espaço eu desejo adicionar.

```

plot_exemplo +
  theme(
    plot.title = element_text(
      face = "italic",
      hjust = 0.5,
      margin = margin(b = 20)
    ),
    legend.title = element_text(face = "bold", size = 26/.pt)
  )

```



Fonte: Elaboração própria do autor.

9.5 Plano de fundo (*background*) e *grid*

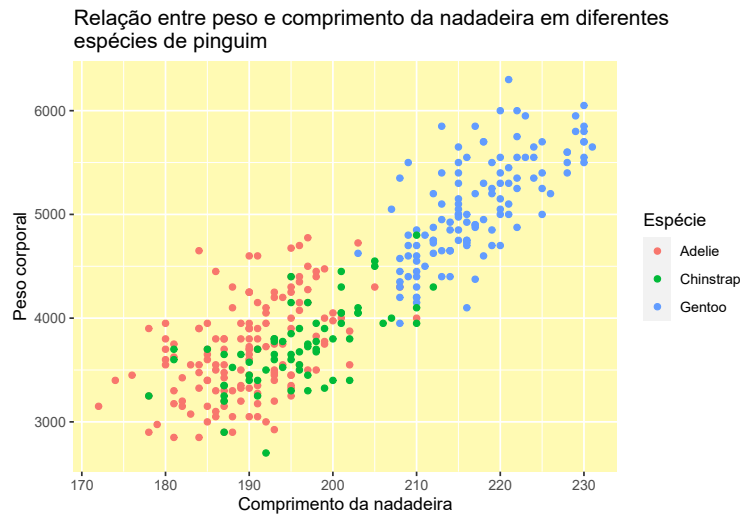
O tema padrão do ggplot pode ser muito esquisito, ou simplesmente “feio” para muita gente. Um de seus elementos que mais recebem críticas, é o plano de fundo do gráfico, que por padrão é colorido de cinza claro. Todos os argumentos de `theme()`, que controlam os elementos que se encontram no plano de fundo, começam por `panel.*`. Você pode, por exemplo, alterar as configurações gerais do plano de fundo pelo argumento `panel.background`, que é associado ao tipo “retângulo” - `element_rect()`.

No exemplo abaixo, estou alterando a cor deste plano de fundo, para uma cor levemente “amarelada”. Lembra quando eu defini que o ggplot trata de forma distinta as formas geométricas de área, onde se você quisesse preencher esta forma com uma cor, você deveria utilizar o argumento `fill`, ao invés de `color`? Aqui a mesma coisa ocorre, pois o plano de fundo do gráfico é associado a um formato de área (retângulo).

Por isso, se utilizar o `color`, você irá colorir apenas as bordas do gráfico, e não preencher o plano de fundo com uma cor. Em ambos argumentos, você pode fornecer um dos nomes de cor que o R consegue reconhecer (por exemplo, “white”, “black”)¹, ou então, você pode fornecer um código HTML dessa cor.

```
plot_exemplo +
  theme(
    panel.background = element_rect(fill = "#fffab3")
  )
```

¹Você pode ver a lista completa de nomes, ao rodar a função `colors()` no console.



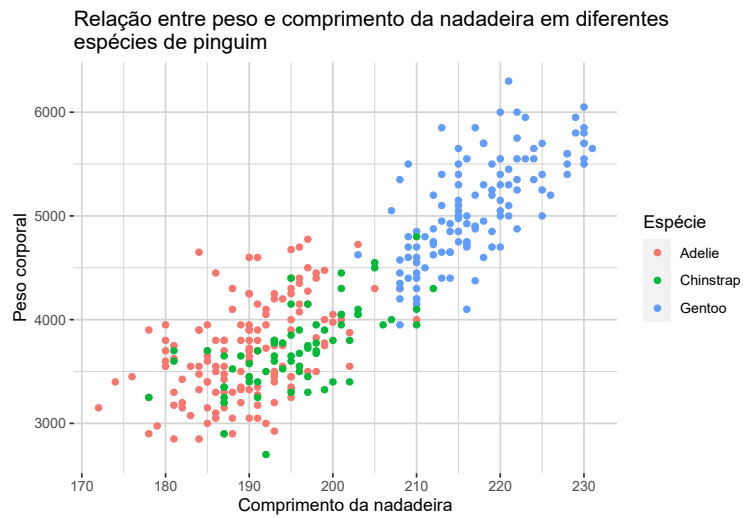
Fonte: Elaboração própria do autor.

Se antes você não gostava do cinza, você provavelmente está gostando menos ainda dessa cor amarelada. Bem, neste caso podemos ficar então com o branco padrão, que está na grande maioria dos gráficos. As linhas do *grid* já estão na cor branca, por isso, podemos colorir também essas linhas para um cor diferente, de modo a mantê-las visíveis.

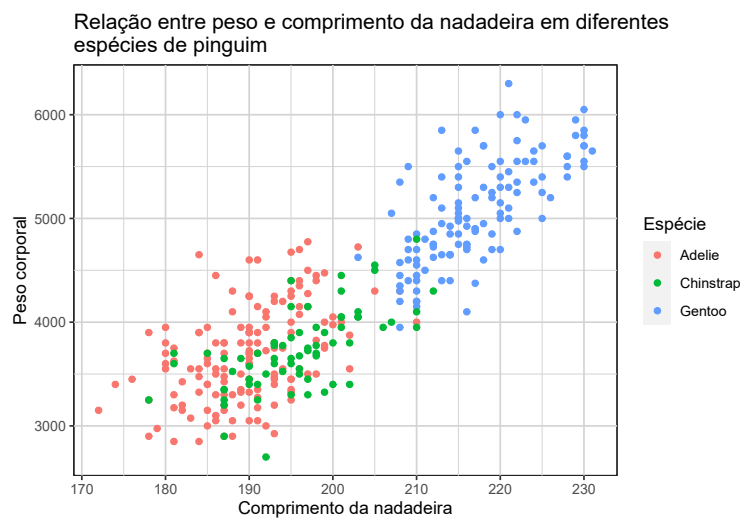
```
plot_exemplo +
  theme(
    panel.background = element_rect(fill = "white"),
    panel.grid = element_line(color = "#d4d4d4")
  )
```

Apesar do gráfico estar agora em um tema mais “padrão”, você talvez ache estranho a forma como as linhas do *grid* estão no momento. Pois elas estão sem um “limite”, ou aparentam estar “invadindo” o espaço de outros elementos do gráfico. Talvez o que você precise, seja marcar a borda do gráfico, para construir uma caixa, e definir estes limites do *grid*. Tudo que você precisa fazer, é usar o `color` em `panel.background`, para colorir essas bordas.

```
plot_exemplo +
  theme(
    panel.background = element_rect(
      fill = "white",
      color = "#222222"
    ),
    panel.grid = element_line(color = "#d4d4d4")
  )
```



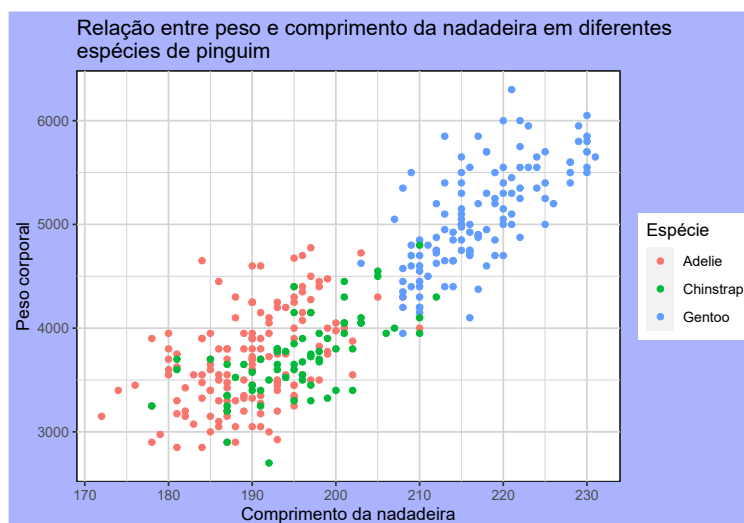
Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

Um outro componente que faz parte do gráfico, é o plano de fundo de toda a área do gráfico. Ou seja, toda a área de sua tela que engloba os títulos, os valores, as legendas e o espaço do gráfico. Essa área é controlada pelo argumento `plot.background`. Não sei por que você faria isso, mas com esse argumento, você pode por exemplo, pintar toda a área do gráfico de azul claro.

```
plot_exemplo +
  theme(
    panel.background = element_rect(
      fill = "white",
      color = "#222222"
    ),
    panel.grid = element_line(color = "#d4d4d4"),
    plot.background = element_rect(fill = "#abb3ff")
  )
```

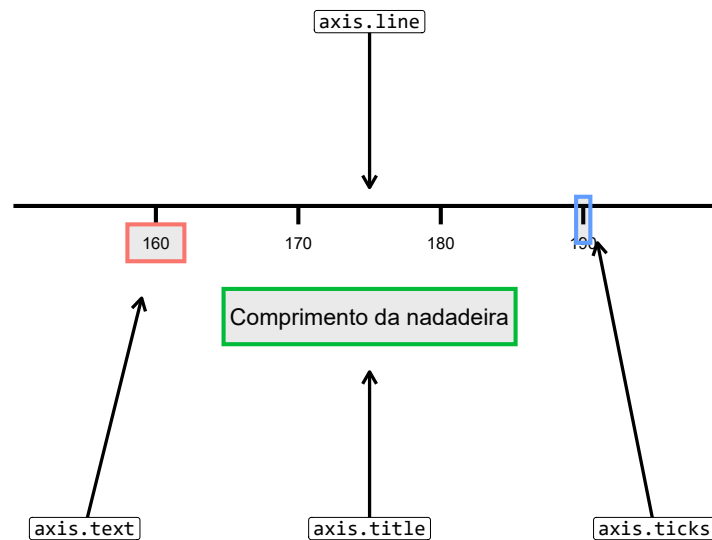


Fonte: Elaboração própria do autor.

9.6 Eixos do gráfico

Todos os elementos que se encontram nos eixos do gráfico, são controlados pelos argumentos de `theme()` que se iniciam por `axis.*`. Você pode ver os argumentos que controlam cada um dos componentes do eixo, pela figura abaixo.

No tema padrão do ggplot, a linha do eixo (`axis.line`) já não aparece. Portanto, se você quiser eliminar completamente um eixo do seu gráfico, você precisa apagar apenas os outros três componentes. Sendo este, um outro motivo de estranhamento de várias pessoas sobre o tema padrão do ggplot. Por isso, talvez seja interessante para você incluir no seu gráfico, as linhas do eixo, e para esse fim, basta redefinir o seu argumento (`axis.line`) com `element_line()`.

Figura 9.2: Elementos que compõe um eixo do gráfico

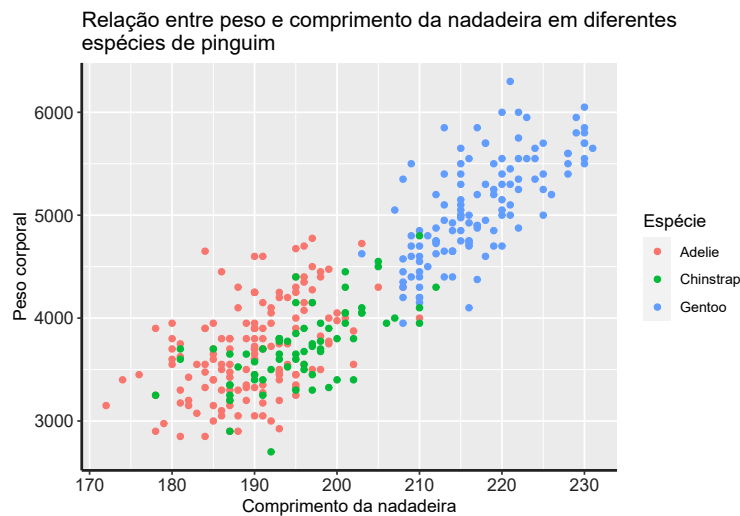
Fonte: Elaboração própria do autor.

Um detalhe muito importante, é que a função `theme()` possui tanto o argumento geral do componente do eixo (e.g. `axis.line`), que afeta ambos os eixos (x e y) ao mesmo tempo, quanto o argumento que afeta os eixos individualmente (e.g. `axis.line.x` e `axis.line.y`). Isso vale para todos os outros três componentes do eixo, e portanto, caso você queira que a modificação afete apenas um dos eixos, você deve utilizar os argumentos que possuem o eixo no nome, ao invés dos argumentos gerais.

Uma configuração que aplico com bastante frequência em meus gráficos, é escurecer os valores do eixo (`axis.text`). Por padrão, os valores vem em um cinza claro, e por causa disso, a leitura desses valores pode ficar muito prejudicada ao exportar esse gráfico, e incluí-lo em um artigo, informativo ou relatório que estou escrevendo. Desse modo, no exemplo abaixo, além de reposicionar as linhas dos eixos, eu também utilizo o argumento `color` em `axis.text`, para colorir esses valores com uma cor mais escura.

Além dessas modificações, para garantir que o meu leitor consiga ler esses números, eu ainda aumento levemente o tamanho dos valores do eixo, pelo argumento `size`. Como eu disse anteriormente, esse argumento trabalha, por padrão, com milímetros. Você pode novamente utilizar a variável `.pt` para transformar esse valor para pontos (pt).

```
plot_exemplo +
  theme(
    axis.line = element_line(size = 0.8, color = "#222222"),
    axis.text = element_text(size = 11, color = "#222222")
  )
```



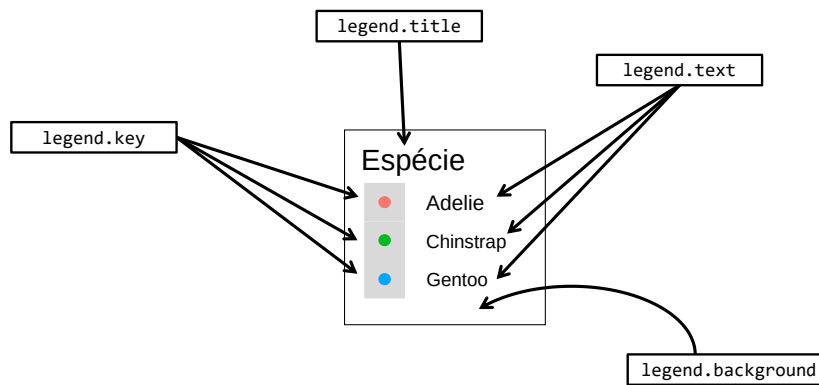
9.7 Configurações temáticas em uma legenda

A legenda de seu gráfico, é um guia que lhe mostra como os elementos visuais percebidos em seu gráfico, se traduzem de volta aos valores observados em sua base de dados. Em outras palavras, é a legenda que mapeia as cores, formas e tamanhos dos elementos de seu gráfico, de volta aos valores apresentados em sua base de dados (WILKINSON, 2005; WICKHAM, 2016). Sem a legenda, nós não sabemos qual o valor que a cor vermelha em nosso gráfico se refere, nem quanto o tamanho de um objeto, representa em nível de uma variável numérica.

Temos na figura 9.3, os componentes de uma legenda em um gráfico do ggplot, e os seus respectivos argumentos em `theme()`. Há outros argumentos relacionados em `theme()`, como `legend.text.align`, `legend.margin` e `legend.position`, que não afetam a temática de algum componente específico da legenda, mas sim, o alinhamento de certos componentes, ou a margem da legenda em relação ao gráfico, ou a posição geral da legenda.

Como exemplo, podemos preencher o plano de fundo da legenda com alguma cor específica em `legend.background` (argumento `fill`), assim como podemos contornar as bordas dessa legenda com alguma outra cor (argumento `color`). Podemos alterar o alinhamento do texto da legenda, ou mais especificamente, os rótulos de cada item da legenda, através de `legend.text.align`, ao fornecermos um número entre 0 (alinhado totalmente à esquerda) e 1 (alinhado totalmente à direita). Também podemos utilizar a função `element_text()` em `legend.title`, para alterarmos a fonte (argumento `family`), o tamanho (argumento `size`) o estilo da fonte (argumento `face`: `bold` - negrito, `italic` - itálico, `bold.italic` - negrito e itálico), e inclusive a cor (argumento `color`) utilizada no título dessa legenda.

Além dessas configurações, possuímos um bom controle sobre a posição da legenda ao longo da área do gráfico, através do argumento `legend.position`. Por padrão, toda legenda gerada pelo

Figura 9.3: Itens que compõe uma legenda

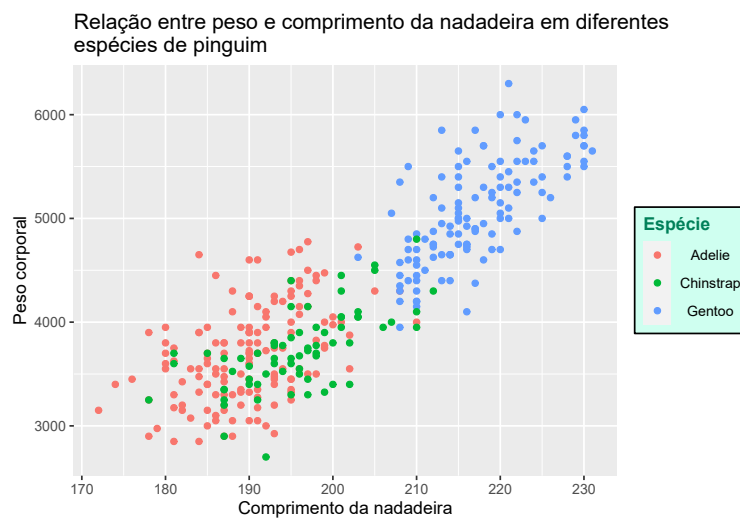
Fonte: Elaboração própria do autor.

ggplot, será posicionada à direita do gráfico, entretanto, esse padrão tende a ocupar muito espaço do gráfico, por isso eu particularmente prefiro posicionar as minhas legendas, na parte inferior do gráfico. Para isso podemos fornecer o valor `bottom` ao argumento. O argumento `legend.position`, aceita outros quatro valores pré-definidos: `top` (topo do gráfico); `left` (esquerda do gráfico); `right` (direita do gráfico); `none` (nenhum local do gráfico).

Você pode utilizar o valor pré-definido `none` em `legend.position`, para eliminar completamente a legenda do gráfico. Isso é uma boa forma de aumentar o espaço do gráfico, porém, você elimina uma fonte importante de informação, portanto, considere com cuidado se as informações dispostas em sua legenda, são irrelevantes para o seu gráfico. Para além das posições pré-definidas, podemos inclusive posicionar a nossa legenda, para dentro do gráfico, através de `legend.position`. Para isso, você precisa fornecer dentro de um vetor, a posição (x, y) no plano cartesiano em que você deseja centralizar a sua legenda, de acordo com um valor entre 0 e 1. Você pode interpretar esse sistema, como percentis da distribuição dos valores presentes no eixo. Ou seja, se você fornecer o vetor `c(0.1, 0.9)`, a legenda será posicionada no 10º percentil da escala do eixo x, e no 90º percentil da escala do eixo y.

```
plot_exemplo + theme(
  legend.background = element_rect(fill = "#cffff0", color = "black"),
  legend.text.align = 0.5,
  legend.title = element_text(face = "bold", color = "#008059"),
)
```

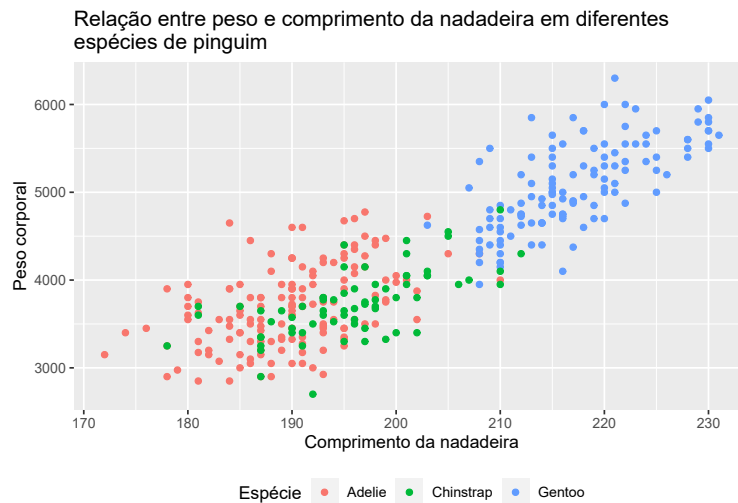
```
## Warning: Removed 2 rows containing missing values (geom_point).
```

Fonte: Elaboração própria do autor.

```
plot_exemplo + theme(  
  legend.position = "bottom"  
)
```

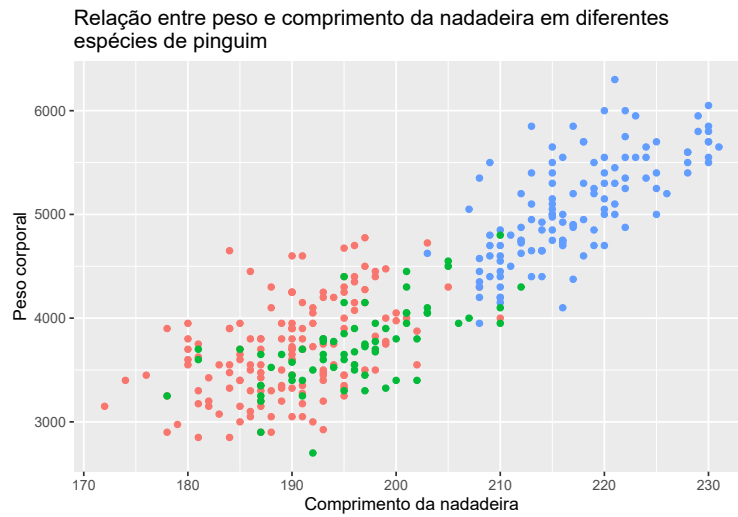
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Fonte: Elaboração própria do autor.

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

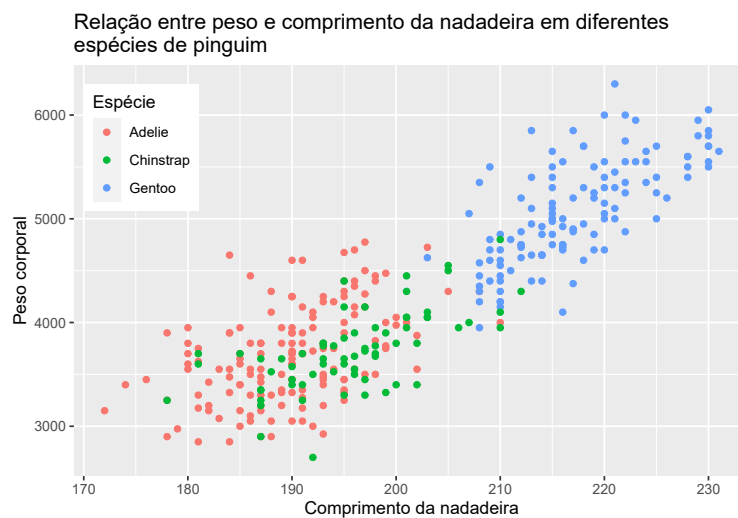
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Fonte: Elaboração própria do autor.

```
plot_exemplo + theme(
  legend.position = c(0.1, 0.8)
)

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



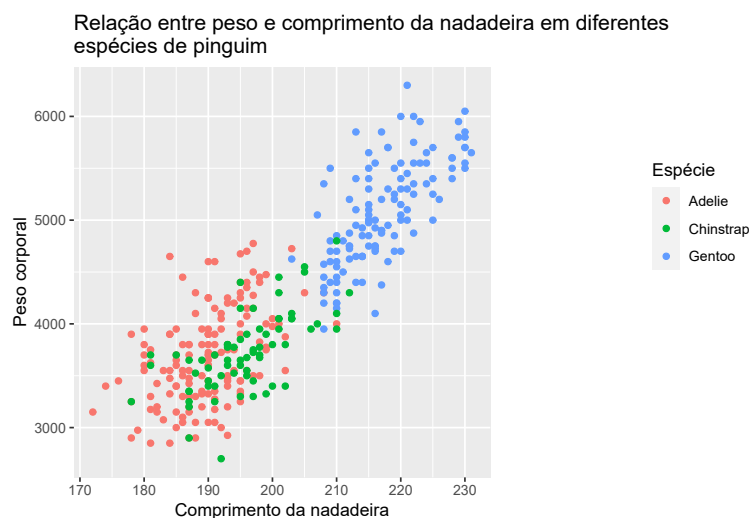
Fonte: Elaboração própria do autor.

Para mais, temos algumas outras configurações possíveis sobre a margem da legenda em relação à área gráfica, através do argumento `legend.margin` e da função `margin()`. Ou seja, nós podemos

afastar a legenda da área do gráfico, ou da base do gráfico. Em outras palavras, nós podemos adicionar espaço na base (b), no topo (t), à direita (r), ou à esquerda (l) da legenda, através da função `margin()`.

```
plot_exemplo + theme(
  legend.margin = margin(l = 90, b = 70)
)

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



Fonte: Elaboração própria do autor.

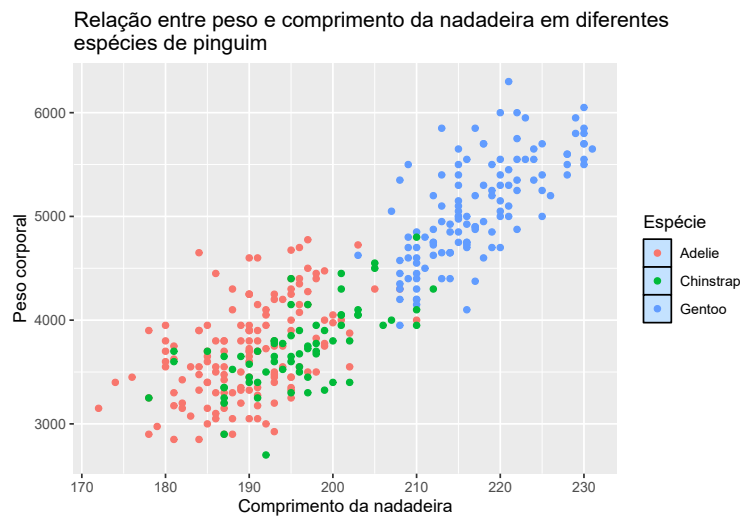
Por último, nós também podemos configurar os itens da legenda, através do argumento `legend.key`. Neste argumento, você possui todas as opções de customização oferecidas pela função `element_rect()`. Além de preencher o plano de fundo dos itens (argumento `fill`), ou de criar uma borda (argumento `color`), também temos a opção de alterar o tamanho desses itens (argumento `size`).

```
plot_exemplo + theme(
  legend.key = element_rect(fill = "#c4e2ff", color = "black")
)

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

9.8 Alterando a temática em facetas

Quando você adiciona facetas a um gráfico, você possui novos elementos que talvez sejam de seu interesse configurá-los. Por exemplo, o título de cada faceta, ou o plano de fundo desse título. Todos



os argumentos de `theme()` que controlam elementos das facetas do gráfico, começam por `strip.*`. No exemplo abaixo, eu estou redefinindo as cores do interior e das bordas do plano de fundo da faceta, além da cor do título da faceta.

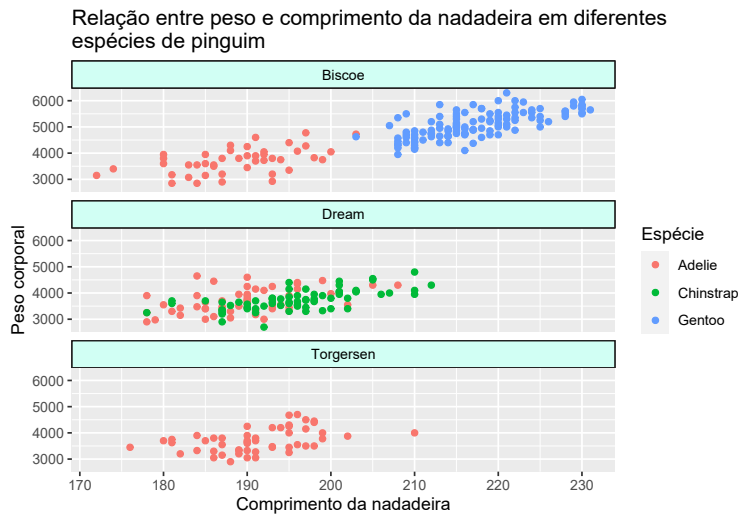
```
plot_exemplo +
  facet_wrap(~island, nrow = 3) +
  theme(
    strip.background = element_rect(color = "#222222", fill = "#d1fff4"),
    strip.text = element_text(color = "black")
  )
```

9.9 Alterando as fontes do seu gráfico

Este é provavelmente o tópico de maior interesse para você ao customizar os seus gráficos, pois você sabe muito bem o potencial impacto que a tipografia pode gerar sobre ele. Eu separei uma seção para discutir apenas esse assunto, pois como você descobrirá bem cedo, inserir fontes de seu sistema (ou fontes customizadas) em seu gráfico pode ser uma dor de cabeça bem grande.

Essa dificuldade ocorre em qualquer programa², linguagem ou sistema que trabalha com diversos *device*'s gráficos, como é o caso do R. Como comentamos na seção **Exportando os seus gráficos do ggplot**, um *device* gráfico é a *engine* que vai gerar o arquivo de imagem, onde o seu gráfico será guardado. Diferentes *engine*'s, geram um arquivo de tipo diferente, como .png, ou .jpeg,

²Caso você queira entrar em mais detalhes, um bom início é o artigo intitulado “Text Rendering Hates You”, de Alexis Beingessner: <https://gankra.github.io/blah/text-hates-you/>



ou .tiff, ou um arquivo .pdf. Ou seja, cada um desses formatos de arquivo, utilizam um *device* gráfico diferente para construir o arquivo que irá guardar o seu gráfico.

Antes de definirmos os problemas existentes, e explicar quais são os processos necessários, para que você possa utilizar qualquer fonte que esteja em sua máquina, em seus gráficos do ggplot. Eu vou mostrar quais são as três opções de fonte, que são garantidas de funcionar em seus gráficos do ggplot, e em qualquer máquina. Essas três opções são:

- 1) sans: Fonte Arial.
- 2) serif: Fonte Times New Roman.
- 3) mono: Fonte Courier New.

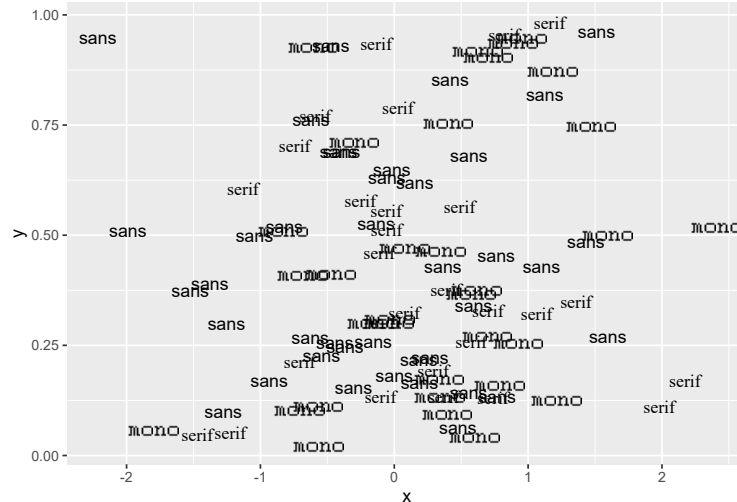
Portanto, em qualquer máquina que você estiver, você pode utilizar um desses três nomes (sans, serif e mono) para se referir a uma dessas três fontes acima, em seu gráfico do ggplot. Se você quer utilizar diferentes fontes ao longo dos dados mostrados em seu gráfico, você deve definir como essas fontes utilizadas, vão variar ao longo do gráfico, através da função `aes()`. Mais especificamente, você deve utilizar o argumento `family` na função `aes()`, dentro das funções que estão desenhando os textos em seu gráfico, de acordo com os dados presentes em sua tabela, como as funções `geom_text()` e `geom_label()`. Ou seja, não estamos falando do tema do gráfico, e sim dos pontos que representam os seus dados no plano cartesiano. Um exemplo de uso dessa ideia, é mostrado logo abaixo.

Neste momento, você deve pensar se você deseja variar as fontes utilizadas ao longo do gráfico, ou se você quer manter ela fixa, ou em outras palavras, que uma mesma fonte seja utilizada em todos os rótulos e textos dispostos no gráfico. Se você quer variar a fonte, você deve criar uma nova variável em sua tabela, contendo os nomes dessas fontes, e em seguida, conectar essa variável ao argumento

family, dentro de aes(). Mas se você quer manter essa fonte fixa, basta fornecer o nome dela à family, fora de aes().

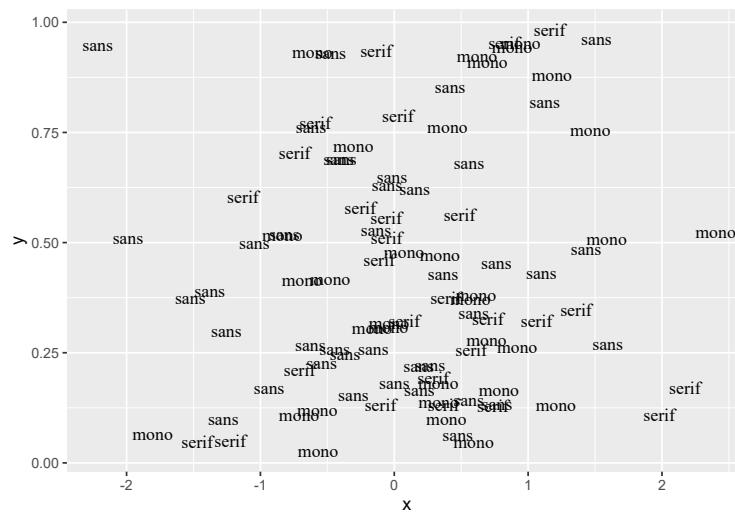
```
set.seed(1)
tab <- data.frame(
  x = rnorm(100),
  y = runif(100),
  fonte = sample(
    c("sans", "serif", "mono"),
    size = 100,
    replace = TRUE
  )
)

### Variar a fonte utilizada ao longo do gráfico
ggplot(tab) +
  geom_text(
    aes(x = x, y = y, family = fonte, label = fonte)
  )
)
```



Fonte: Elaboração própria do autor.

```
### Ou manter a fonte fixa ao longo de todo o gráfico
ggplot(tab) +
  geom_text(
    aes(x = x, y = y, label = fonte),
    family = "serif"
  )
)
```

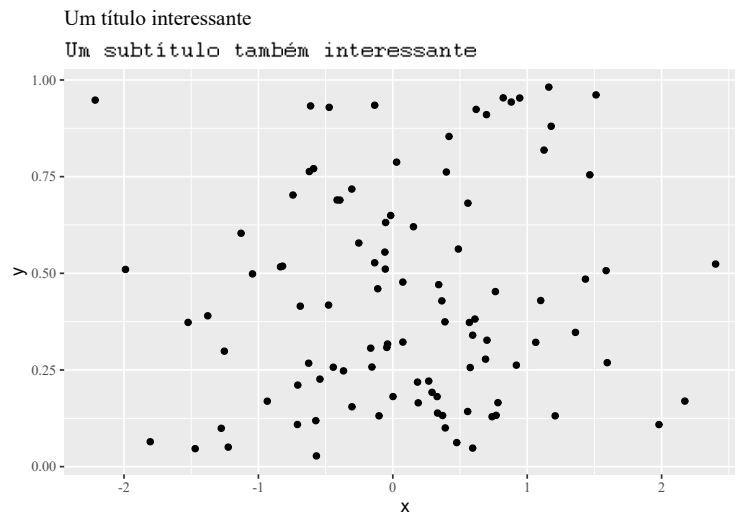


Fonte: Elaboração própria do autor.

Portanto, é dessa forma que podemos definir a fonte utilizada nas funções geom que representam os nossos dados no gráfico. Entretanto, para alterarmos a fonte em elementos temáticos (elementos que não dizem respeito, ou que não estão diretamente conectados com os seus dados) do gráfico, essas configurações devem ser realizadas dentro da função theme. Basta utilizarmos o argumento family presente em element_text(), e definirmos o nome da fonte que desejamos empregar.

Um atalho útil, caso você deseja utilizar a mesma fonte em todos os elementos temáticos do gráfico, se trata do argumento text (que se refere a todos os elementos temáticos do tipo “texto”) na função theme(), e definir com a função element_text() a fonte utilizada. Ou seja, basta adicionar a seguinte estrutura dentro de theme(): text = element_text(family = <fonte>). Porém, caso você deseja utilizar uma fonte diferente em cada componente temático do gráfico, você obrigatoriamente deve definir separadamente a fonte a ser utilizada, em cada argumento de theme() que corresponde a esses componentes estéticos.

```
ggplot(tab) +
  geom_point(aes(x = x, y = y)) +
  labs(
    title = "Um título interessante",
    subtitle = "Um subtítulo também interessante"
  ) +
  theme(
    plot.title = element_text(family = "serif"),
    plot.subtitle = element_text(family = "mono"),
    axis.text = element_text(family = "serif")
  )
```

Fonte: Elaboração própria do autor.

9.9.1 Importando novas fontes para o R

“At its core text and fonts are just very messy, with differences between operating systems and font file formats to name some of the challenges”. (PEDERSEN, 2020).

Agora que vimos como implementar o que o ggplot2 do R oferecem já de “fábrica” ao usuário, vou explicar como podemos expandir para as demais fontes presentes em sua máquina. Para isso, você irá precisar de pacotes que facilitam esse processo, sendo o principal deles, o *extrafont*. É importante destacar, que o métodos que vou explicar aqui, permite o uso apenas de fontes *TrueType*, ou em outras palavras, fontes onde o seu arquivo possui a extensão *.ttf*. Tendo essas considerações em mente, se você não possui este pacote instalado no seu computador, você deve rodar o comando abaixo.

```
install.packages("extrafont")
```

Uma das dificuldades no uso de diferentes fontes no R, é encontrar os arquivos dessas fontes. Pois a forma e o local em que os arquivos dessas fontes estão guardados, varia ao longo dos sistemas operacionais. Além disso, também não há um padrão definido no nome desses arquivos. Mais especificamente, existe uma forma de interpretarmos as classes e famílias de cada fonte, porém, não há um padrão muito bem definido de como os arquivos dessas fontes deveriam ser nomeados para tal processo. Logo, os nomes dos arquivos dessa fonte, podem gerar incongruências e conflitos com os arquivos de outras fontes, e com isso, o R talvez não consiga diferenciar uma fonte da outra.

Um outro grande problema, está no fato de que cada *device* gráfico, possui em geral, exigências diferente quanto aos arquivos dessas fontes. Por exemplo, as *engines* que produzem arquivos PDF, precisam obrigatoriamente de um arquivo *.afm* (*Adobe Font Metrics File*) para cada fonte utilizada, e você muito provavelmente não possui tal arquivo (QIU, 2015).

Esses dois problemas podem ser resolvidos com o uso das funções provenientes do pacote `extrafont`. Tendo isso em mente, a primeira coisa que você deve fazer, **sempre** que for definir a localização de uma nova fonte que você acabou de baixar da internet, ou de uma fonte que já está instalada no seu sistema operacional, é **reiniciar o R**. No RStudio, você possui o atalho `Ctrl + Shift + F10`, ou então se preferir, você pode ir à aba `Session`, e escolher a opção `Restart R`.

Muitas vezes o processo que vamos executar a seguir, falha de alguma forma caso você já tenha outros pacotes conectados a sua sessão atual. Devido a isso, é importante que você inicie o processo com uma sessão limpa. Após reiniciar o R, chame pelo pacote `extrafont` com a função `library()`.

```
library(extrafont)
```

```
## Registering fonts with R
```

O que vamos fazer a seguir, é “importar” as fontes para uma base de dados, ou dito de outra forma, vamos guardar a localização dos arquivos dessas fontes, em um local que seja de fácil acesso ao R. Logo, o papel que o pacote `extrafont` vai desempenhar, será construir uma planilha onde ele irá guardar a localização desses arquivos, e diversas outras informações como o nome e a classe de cada fonte. Assim, sempre que você precisar dessa fonte e chamar por ela, o R irá procurar pela localização dos arquivos dessa fonte, nessa base de dados criada por `extrafont`.

Para executar esse passo, você deve utilizar a função `font_import()`. Essa função procura automaticamente pelas fontes instaladas em seu sistema operacional. Eu trabalho com o Windows, que possui uma pasta específica onde ele guarda os arquivos de cada uma dessas fontes instaladas. Logo, ao rodar a função `font_import()` no Windows, ela irá procurar automaticamente por essa pasta. Mas caso a função, por algum motivo não estiver encontrando essa pasta, você pode tentar definir o caminho até essa pasta, direcionando assim a função. No caso do Windows, essa pasta fica na localização de seu computador definida abaixo.

```
### Pasta do Windows que contém  
### as fontes instaladas em seu sistema  
C:\Windows\Fonts
```

```
### Basta fornecer este endereço no  
### argumento paths de font_import()  
font_import(paths = "C:/Windows/Fonts")
```

Ao rodar a função `font_import()` no console, ela irá lhe mostrar a mensagem abaixo, perguntando se você deseja continuar o processo. Para continuar, basta enviar para o console, a letra “y”. A partir daí, a função irá iniciar o processo, encontrando todas as fontes disponíveis em seu sistema, e guardando as informações dessas fontes.

```
Importing fonts may take a few minutes, depending on the number  
of fonts and the speed of the system.  
Continue? [y/n]
```

Ao terminar o processo, você terá definido a localização dos arquivos e coletado as informações necessárias da fonte, e você não precisará realizar novamente este processo, pois essas informações estão salvas na base de dados criada por `extrafont`. Você irá rodar novamente a função `font_import()`, apenas no caso em que você quiser adicionar uma nova fonte, que não estava instalada anteriormente em seu sistema.

Apesar de já muito útil, você talvez queira mudar o comportamento da função `font_import()`, que vai procurar pelos arquivos presentes apenas na pasta principal de fontes de seu sistema operacional. Por exemplo, talvez você queira importar uma fonte que ainda não está instalada em seu computador, por exemplo, uma fonte que você acabou de baixar do Google Fonts.

Para essas ocasiões, eu recomendo que você instale essas fontes em seu sistema operacional, antes de prosseguir para os próximos passos. Após a instalação, você pode utilizar novamente o argumento `paths` da função, onde você pode definir a pasta na qual a função irá procurar pelos arquivos dessas fontes (arquivos com extensão `.ttf`) que você acaba de instalar. Eu no caso, recomendo que você crie uma pasta, e guarde nela todos os arquivos de fontes que você deseja importar. Por exemplo, eu tenho uma pasta onde guardo todas as fontes que baixo do Google Fonts, e portanto, caso eu queira importar uma nova fonte que eu acabei de baixar, para o R, eu coloco os arquivos dessa fonte dentro dessa pasta, e forneço o endereço dessa pasta para `font_import()`.

```
font_import(paths = "C:/Users/Pedro/Downloads/Google Fonts")
```

Portanto, após esse passo, onde importamos essas fontes para o R, as fontes ainda não estão disponíveis para serem utilizadas no `ggplot`. Você até o momento, guardou as informações necessárias dessas fontes, mas ainda não forneceu essas informações ao R. Por isso, para utilizar as fontes que você salvou, você deve rodar a função `loadfonts()`, para “carregar” essas fontes para a sua sessão atual do R. É importante também destacar, que você deve **sempre** rodar essa função antes mesmo de chamar pelo pacote `ggplot2`, para evitar *bugs* indesejados.

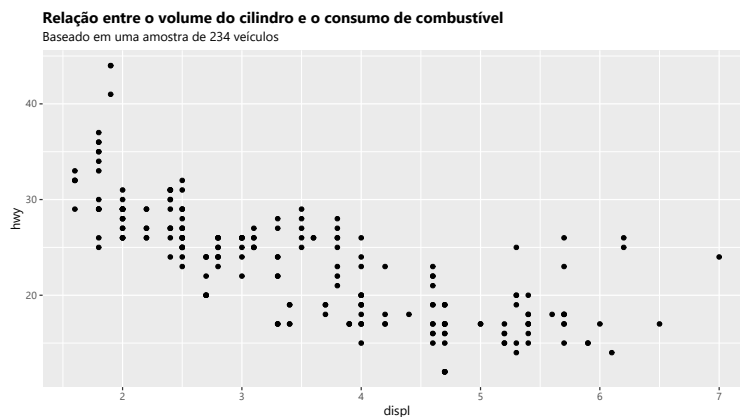
Ou seja, em toda sessão no R, em que você estiver gerando um gráfico do `ggplot`, e deseja utilizar alguma fonte que já esteja registrada na planilha de `extrafont`, você muito provavelmente terá que reiniciar o R, chamar pelo pacote `extrafont`, e carregar as fontes salvas pela função `loadfonts()`, antes mesmo de chamar pelo pacote `ggplot2` e de recriar o seu gráfico. Apesar deste processo não ser sempre necessário, ele é em geral a opção mais segura. Após esse passo, você pode utilizar a fonte desejada normalmente em seu gráfico de `ggplot`, basta se referir a ela pelo seu nome nos argumentos `family`.

```
library(extrafont)
loadfonts()
```

```
library(ggplot2)
```

```
### O meu gráfico
ggplot(mpg) +
  geom_point(aes(x = displ, y = hwy)) +
```

```
theme(  
  text = element_text(family = "Segoe UI"),  
  plot.title = element_text(face = "bold")  
) +  
labs(  
  title = "Relação entre o volume do cilindro e o consumo de combustível",  
  subtitle = "Baseado em uma amostra de 234 veículos"  
)
```



Fonte: Elaboração própria do autor.

Caso você não se lembre do nome exato da fonte que deseja, ao executar a função `fonts()`, você pode acessar uma lista que contém os nomes de todas as fontes que foram importadas para a sua sessão - através da função `loadfonts()`, e que portanto, estão disponíveis para uso em sua sessão.

Depois desses passos, ao conseguir gerar o gráfico que você queria, utilizando as fontes que você desejava, será provavelmente de seu desejo, salvar esse gráfico, exportá-lo para algum arquivo. Você pode fazer isso normalmente pelos métodos que mostramos na seção [Exportando os seus gráficos do ggplot](#), especialmente se você escolher exportar o seu gráfico, para algum tipo de arquivo *bitmap*, ou uma imagem do tipo *raster* (arquivos PNG, JPEG/JPG, BMP ou TIFF). Porém, se você escolheu exportar o seu gráfico para um arquivo PDF, você talvez enfrente alguns problemas, como as suas fontes desaparecerem do resultado!. Nós mostramos na seção [Arquivos PDF e SVG](#), como resolver esse problema, que envolve o uso da função `cairo_pdf()`.

Exercícios

Exercício 1

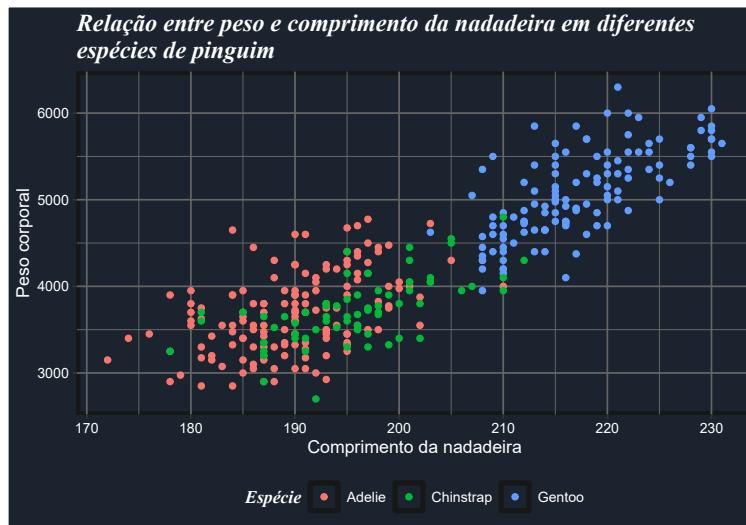
Voltando ao gráfico salvo no objeto `plot_exemplo`, o qual utilizamos ao longo de todo este capítulo. Seu objetivo nessa questão é criar um objeto `tema` que seja capaz de aproximar `plot_exemplo` o máximo possível do gráfico abaixo.

```
library(ggplot2)
library(palmerpenguins)

plot_exemplo <- ggplot(data = penguins) +
  geom_point(
    aes(
      x = flipper_length_mm,
      y = body_mass_g,
      color = species)
  ) +
  labs(
    title = "Relação entre peso e comprimento da nadadeira
em diferentes\nespécies de pinguim",
    x = "Comprimento da nadadeira",
    y = "Peso corporal",
    color = "Espécie"
  )

tema <- theme(
  # Coloque as especificações necessárias
  # para que plot_exemplo se torne
  # o gráfico abaixo
)

plot_exemplo + tema
```



Fonte: Elaboração própria do autor.

Exercício 2

Em cada item abaixo, vamos utilizar novamente o gráfico salvo no objeto `plot_exemplo`. Esses itens vão lhe questionar sobre algum erro específico, ou, lhe requisitar.

- A) Faça a legenda do gráfico `plot_exemplo` desaparecer.
- B) Identifique porque o erro abaixo ocorre, e tente corrigi-lo.

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

Erro: Only elements of the same class can be merged
Run ``rlang::last_error()`` to see where the error occurred.

- C) Contorne a área do *grid* (isto é, a área em que as formas geométricas do gráfico são desenhadas pela função `geom_*()`) por uma linha de cor `"#222222"`.

Capítulo 10

Manipulação e transformação de *strings* com *stringr*

10.1 Introdução e pré-requisitos

Neste capítulo, vamos aprender mais sobre operações especializadas em dados textuais (dados do tipo `character`), ou como são mais comumente denominados em programação, *strings*. Esse capítulo também oferece uma introdução a um dos principais e mais importantes tópicos em processamento de texto, as expressões regulares (*regular expression*), ou *regex* como é mais conhecida. Para aplicarmos as diversas operações expostas, vamos utilizar as funções disponíveis no pacote *stringr*. Esse pacote está incluso no *tidyverse*, logo, para ter acesso às funções apresentadas, você pode chamar pelo *tidyverse* ou pacote *stringr* diretamente, por meio do comando `library()`.

```
library(stringr)
library(tidyverse)
```

10.2 Algumas noções básicas

Textos ou *strings* no R, são criados ao contornarmos um determinado texto por aspas (duplas - `"`, ou simples - `'`), e cada letra, espaço, símbolo ou número que compõe esse texto, é comumente denominado de caractere. Caso você se esqueça de fechar o par de aspas que contorna o seu texto, o R vai esperar até que você complete a expressão. Ou seja, em seu console, estaria acontecendo algo parecido com o que está abaixo. Lembre-se que você pode apertar a tecla `Esc`, para abortar a operação, caso você não consiga completá-la.

```
> x <- "Olá eu sou Pedro!
+
+
```

Como as aspas são responsáveis por delimitar esse tipo de dado, para que você possa incluir esse caractere em alguma cadeia de texto, você tem duas alternativas: 1) se você está contornando o texto com aspas duplas, utilize aspas simples, ou vice-versa; 2) contornar o comportamento especial das aspas, ao posicionar uma barra inclinada a esquerda antes de cada aspa (`\"` ou `\'`).

```
"Olá! Esse é um texto qualquer"
```

```
## [1] "Olá! Esse é um texto qualquer"
```

```
"Para incluir aspas (') em um string"
```

```
## [1] "Para incluir aspas (') em um string"
```

```
"Será que \"alienígenas\" existem de fato?"
```

```
## [1] "Será que \"alienígenas\" existem de fato?"
```

Além disso, textos podem incluir diversos outros caracteres especiais. Sendo os principais exemplos, os caracteres de tabulação (`\t`), e de quebra de linha (`\n`). Entretanto, uma quantidade muito grande desses caracteres especiais, podem dificultar a nossa compreensão do conteúdo presente em um texto. Logo, há vários momentos em que desejamos visualizar o texto representado em um *string* de maneira “crua”. Para isso, podemos aplicar a função `writeLines()` sobre o texto em questão.

```
texto <- "Receita:\n\t\t2 ovos\n\t\t3 copos e meio de farinha
\t\t2 copos de achocolatado\n\t\t1 copo de açúcar\n\t\tMeio copo de óleo
\t\t1 colher (de sopa) de fermento
\t\t1 colher (de café) de bicabornato de sódio\n\t\t..."
writeLines(texto)
```

```
## Receita:
##      2 ovos
##      3 copos e meio de farinha
##      2 copos de achocolatado
##      1 copo de açúcar
##      Meio copo de óleo
##      1 colher (de sopa) de fermento
##      1 colher (de café) de bicabornato de sódio
##      ...
```

```
texto <- "Será que \"alienígenas\" existem de fato?"
writeLines(texto)
```

```
## Será que "alienígenas" existem de fato?
```


Outro exemplo clássico de caracteres especiais, que são muito encontrados em páginas da internet (e.g. dados coletados em operações de *web scrapping*), são os códigos hexadecimais ou *code points* correspondentes a uma determinada letra presente no sistema Unicode. Descrevemos brevemente na seção **Um pouco sobre fontes, encoding e tipografia**, a importância do Unicode para a universalização dos sistemas de *encoding*, e consequentemente, para a internacionalização de conteúdo.

Cada caractere no sistema Unicode, é representado por um *unicode code point* (HARALAMBOUS, 2007). Em resumo, um *code point* é um número inteiro que pode identificar unicamente um caractere presente no sistema Unicode. Porém, caracteres que são codificados nesse sistema, são normalmente representados pelo código hexadecimal que equivale ao seu respectivo *code point*. Logo, ao invés de um número específico, você normalmente irá encontrar em *strings*, códigos que se iniciam por \u, ou \U, ou ainda U+, seguidos por uma combinação específica de letras e números. Como exemplo, os códigos hexadecimais abaixo equivalem aos *code points* que formam a palavra “Arigatōgozaimashita”, ou “Muito obrigado” em japonês.

```
x <- "\u3042\u308a\u304c\u3068\u3046\u3054\u3056\u3044\u307e\u3057\u305f"
```

Um outro ponto muito importante em *strings* está no uso de barras inclinadas à esquerda. Nós já vimos na seção **Definindo endereços do disco rígido no R**, que para representarmos uma barra inclinada à esquerda em um *string* do R, precisarmos duplicar essa barra. Logo, em *strings*, a sequência \\ significa para o R \. Existem alguns comandos e caracteres especiais que não requerem essa prática, como o comando que forma um *Unicode code point* (como demonstrado acima), que sempre se inicia por uma letra “u” antecedida por uma barra inclinada à esquerda (ex: \u3042). Um outro exemplo são os comandos para tabulações e quebra de linha que acabamos de mostrar (\t e \n). Entretanto, essas exceções são a minoria. Portanto, tenha esse cuidado ao utilizar barras inclinadas à esquerda em seus *strings*.

10.3 Concatenando ou combinando *strings* com paste() e str_c()

Concatenar, significa unir diferentes valores. Porém, essa união pode ocorrer de diferentes maneiras, e como ela ocorre, tende a depender das funções que você utiliza, como você as configura, e com quais tipos de estruturas você está trabalhando. Com isso, eu quero destacar, que o termo concatenar, pode se referir a muitas coisas (ou operações) diferentes. Na linguagem R, uma das principais operações de concatenação está presente na formação de vetores atômicos, mais especificamente, no uso da função c() (abreviação para *combine*), que introduzimos na seção de **Vetores**.

O papel da função c() é criar uma sequência a partir de um conjunto de valores. Essa sequência de valores, é o que forma um vetor, e é o que estabelece uma relação de dependência ou de união entre esses valores, pois os torna parte de uma mesma estrutura. Cada um deles possuem uma ordem, ou uma posição dentro dessa sequência, mas nenhum deles é capaz de gerar essa sequência sozinho.

Entretanto, ao concatenarmos textos, nós geralmente estamos nos referindo a uma operação um pouco diferente. Tradicionalmente, ao concatenarmos um conjunto de textos, nós já possuímos um

vetor (ou mais vetores) em nossas mãos, e desejamos unir cada elemento, ou cada texto contido nesse vetor, de alguma forma lógica. Dentre os pacotes básicos do R, a principal função que realiza esse tipo de operação, é a função `paste()`. Um detalhe importante sobre essa função, é que ela converte, por padrão, qualquer tipo de *input* que você fornecer a ela, para o tipo `character`. Logo, você pode incluir dados numéricos ou de qualquer outro tipo nos *input*'s dessa função.

A forma como a função `paste()` realiza essa união entre os textos, depende diretamente de como você configura os argumentos da função, `sep` e `collapse`, e de quais *input*'s você fornece à função. Se você está fornecendo um único *input* à função, é certo que você está preocupado apenas com o argumento `collapse` (em outras palavras, `sep` é irrelevante nesse caso). Em resumo, o argumento `collapse` define qual o texto que irá separar os diferentes elementos do *input* que você forneceu a função. Em outras palavras, se o *input* que fornecemos é, por exemplo, um vetor de textos, ao definirmos o argumento `collapse`, estamos pedindo à `paste()` que junte todos os diferentes elementos do vetor, dentro de um único texto, separando-os pelo texto que você definiu no argumento `collapse`.

Por exemplo, se eu possuo o vetor `vec` abaixo, e utilizo a função `paste()` sobre ele, veja o que ocorre ao definirmos o argumento `collapse`. Perceba no exemplo abaixo, que todos os elementos do vetor `vec`, foram unidos dentro de um mesmo texto, onde cada um desses elementos são separados pelo texto " : " que definimos no argumento `collapse`.

```
vec <- c("a", "b", "c", "d", "e")

conc_vec <- paste(vec, collapse = " : ")

conc_vec

## [1] "a : b : c : d : e"

## -----
## Um outro exemplo:
nomes <- c("Ana", "Fabrício", "Eduardo", "Mônica")

mensagem <- paste(nomes, collapse = " e ")

mensagem

## [1] "Ana e Fabrício e Eduardo e Mônica"
```

Portanto, o texto que você define em `collapse`, será o texto que vai separar cada um dos elementos do vetor que você fornece como *input* à função `paste()`. Por padrão, o argumento `collapse` é setado para nulo (`NULL`). Isso significa, que se você não definir algum texto para o argumento `collapse`, nada acontece ao aplicarmos a função `paste()` sobre o vetor. Como o argumento `sep` é irrelevante para um único *input*, se você não está interessado nesta operação que ocorre ao definirmos `collapse`, a função `paste()` não é o que você está procurando.

Por outro lado, se você está fornecendo dois ou mais *inputs* à função `paste()`, é provável que você esteja interessado em definir apenas o argumento `sep`, apesar de que o argumento `collapse` pode também ser útil para o seu caso. Ao fornecermos dois ou mais vetores como *inputs*, a função `paste()`, por padrão, tenta unir os elementos desses vetores, de forma a produzir um novo vetor de texto. Por exemplo, se eu forneço dois vetores à função `paste()`, como os vetores `vec` e `id` abaixo, o primeiro elemento do vetor resultante de `paste()` vai possuir os textos presentes no primeiro elemento de ambos os vetores.

```
id <- 1:5
vec <- c("a", "b", "c", "d", "e")

conc_vec <- paste(id, vec)

conc_vec

## [1] "1 a" "2 b" "3 c" "4 d" "5 e"
```

O argumento `sep` é responsável por definir o texto que vai separar os valores de diferentes *input*'s da função `paste()`. Perceba no exemplo acima, os valores dos vetores `id` e `vec`, estão todos separados por um espaço em branco. Isso significa, que por padrão, o argumento `sep` é configurado como um espaço em branco (" "), e portanto, você não precisa definir o argumento `sep`, caso você deseja separar esses valores por um espaço. Mas se há interesse em um texto diferente, para separar esses valores, você deve defini-lo através do argumento `sep`. Por exemplo, você talvez deseja que não haja espaço algum entre esses valores, como exemplo abaixo.

```
id <- 1:5
vec <- c("a", "b", "c", "d", "e")

conc_vec <- paste(id, vec, sep = "")

conc_vec

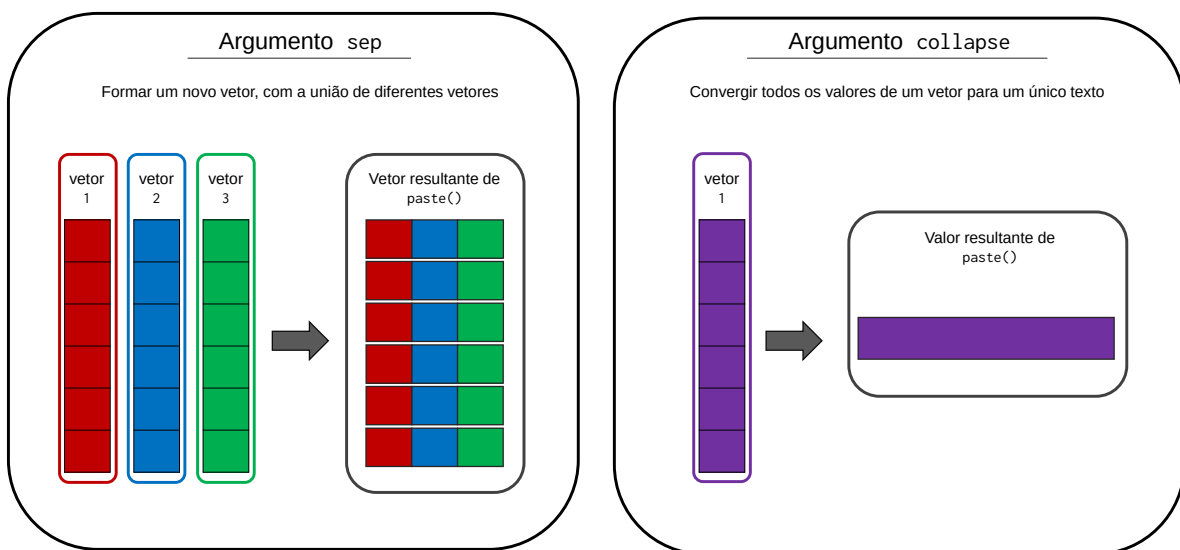
## [1] "1a" "2b" "3c" "4d" "5e"
```

Assim sendo, em uma representação visual, podemos identificar os papéis dos argumentos `sep` e `collapse` da forma apresentada na figura 10.1.

Porém, na maioria das aplicações práticas dessa função, pelo menos um dos *input*'s fornecidos será constante. Por exemplo, uma situação muito comum de uso dessa função, é a construção de caminhos (ou *paths*) para diferentes arquivos. Essa é exatamente a aplicação que utilizamos na seção **Um estudo de caso: uma demanda real sobre a distribuição de ICMS**.

Nessa seção, em uma das primeiras etapas descritas, precisávamos ler ao todo 12 planilhas diferentes, e como descrevemos no capítulo 3, para importarmos qualquer arquivo, nós precisamos fornecer

Figura 10.1: Resumo dos papéis desempenhados pelos argumentos `sep` e `collapse` em `paste()`



Fonte: Elaboração própria do autor.

o caminho até esse arquivo para o R. Com isso, teríamos a tarefa tediosa de construirmos 12 caminhos diferentes (imagine se fossem 36, ou 320 planilhas diferentes a serem lidas). Porém, como todas essas planilhas se encontravam dentro do mesmo diretório de meu computador, eu me aproveitei dessa regularidade, para fabricar esses caminhos de uma maneira prática, através da função `paste()`.

Lembre-se, que inicialmente tínhamos apenas os nomes dessas planilhas contidos no objeto `planilhas` (que está replicado abaixo).

```
planilhas <- list.files("planilhas/")
```

```
planilhas
```

```
## [1] "Abril_2019.xlsx"      "Agosto_2019.xlsx"    "Dezembro_2019.xlsx"
## [4] "Fevereiro_2019.xlsx" "Janeiro_2019.xlsx"   "Julho_2019.xlsx"
## [7] "Junho_2019.xlsx"     "Maio_2019.xlsx"      "Marco_2019.xlsx"
## [10] "Novembro_2019.xlsx"  "Outubro_2019.xlsx"   "Setembro_2019.xlsx"
```

Para criarmos o endereço até cada uma dessas planilhas, necessitávamos juntar o caminho até o diretório em que elas se encontravam ("`planilhas/`"), ao seus nomes. Com isso, podemos utilizar a função `paste()` da seguinte maneira. Perceba que dois *input*'s foram fornecidos a função: o primeiro, consiste apenas no texto "`planilhas/`"; o segundo, são os nomes das planilhas contidos no objeto `planilhas`. Além disso, repare que pelo fato de que o texto "`planilhas/`" ser "constante", `paste()` acaba replicando ele para todos os 12 nomes presentes no objeto `planilhas`.

```
caminhos <- paste("planilhas/", planilhas, sep = "")
```

```
caminhos
```

```
## [1] "planilhas/Abril_2019.xlsx"    "planilhas/Agosto_2019.xlsx"
## [3] "planilhas/Dezembro_2019.xlsx" "planilhas/Fevereiro_2019.xlsx"
## [5] "planilhas/Janeiro_2019.xlsx"  "planilhas/Julho_2019.xlsx"
## [7] "planilhas/Junho_2019.xlsx"    "planilhas/Maio_2019.xlsx"
## [9] "planilhas/Marco_2019.xlsx"    "planilhas/Novembro_2019.xlsx"
## [11] "planilhas/Outubro_2019.xlsx"  "planilhas/Setembro_2019.xlsx"
```

Você talvez tenha percebido, especialmente durante o capítulo 4, que temos uma variante da função `paste()`, chamada `paste0()`. Essa irmã, nada mais é do que um atalho para a função `paste()`, que utiliza por padrão, a configuração `sep = .` Ou seja, em todas as ocasiões em que você estiver concatenando textos de diferentes *input*'s com a função `paste()`, e deseja utilizar nenhum espaço como separador entre os valores de cada *input*, você pode rapidamente executar essa ação por meio da função `paste0()`.

```
caminhos <- paste0("planilhas/", planilhas)
```

```
caminhos
```

```
## [1] "planilhas/Abril_2019.xlsx"      "planilhas/Agosto_2019.xlsx"
## [3] "planilhas/Dezembro_2019.xlsx"  "planilhas/Fevereiro_2019.xlsx"
## [5] "planilhas/Janeiro_2019.xlsx"   "planilhas/Julho_2019.xlsx"
## [7] "planilhas/Junho_2019.xlsx"     "planilhas/Maio_2019.xlsx"
## [9] "planilhas/Março_2019.xlsx"     "planilhas/Novembro_2019.xlsx"
## [11] "planilhas/Outubro_2019.xlsx"   "planilhas/Setembro_2019.xlsx"
```

10.3.1 A função str_c() como uma alternativa para concatenação de strings

Por ser uma operação muito comum e útil, o pacote stringr nos oferece a função str_c(), como uma alternativa à função paste(). Suas diferenças se restringem a dois pontos. Primeiro, a função str_c() foi escrita em C++, e consegue hoje, atingir uma maior eficiência se comparada a função paste(), como demonstrado abaixo. Logo, str_c() pode oferecer uma vantagem considerável, caso você esteja trabalhando com um grande conjunto de textos.

```
library(stringr)
library(microbenchmark)
texto <- sample(letters, size = 1e6, replace = TRUE)

microbenchmark(
  paste(texto, collapse = ""),
  str_c(texto, collapse = "")
)
```

Unit: milliseconds

	expr	min	lq	mean
	paste(texto, collapse = "")	104.7202	107.8384	124.43956
	str_c(texto, collapse = "")	26.3803	26.9155	28.33062
	median	uq	max	neval
	115.8264	129.90345	277.5362	100
	27.1933	29.02705	33.3686	100

Segundo, temos também uma diferença importante sobre as configurações nativas utilizadas por essas funções. Pois a função str_c() adota sep = "" como a sua configuração padrão para o argumento sep (se igualando assim, à função paste0()), ao invés de sep = " ", que é o padrão adotado por paste(). Veja um exemplo abaixo.

```
str_c("Dia", 1:7)

## [1] "Dia1" "Dia2" "Dia3" "Dia4" "Dia5" "Dia6" "Dia7"
```

```
str_c("Dia", 1:7, sep = " ")

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

```
str_c("Dia", 1:7, collapse = "-")
```

```
## [1] "Dia1-Dia2-Dia3-Dia4-Dia5-Dia6-Dia7"
```

Para além dessas diferenças, a função `str_c()` se comporta exatamente da mesma maneira que a função `paste()`. Por isso, pode ser interessante que você adote essa função como o seu padrão para concatenação de textos, especialmente levando-se em conta, a sua maior eficiência.

10.4 Vantagens do pacote stringr

Os pacotes básicos da linguagem R oferecem algumas ferramentas para trabalharmos com *strings*, como a função `paste()` e a família `grep()`. Porém, essas ferramentas são em grande parte, inconsistentes em seus nomes e formas e, por isso, são mais difíceis de se lembrar. Mesmo com essa consideração, eu decidi mostrar a função `paste()` na seção anterior, pelo fato de que ela continua sendo uma função extremamente popular, e que você irá encontrar em todo lugar.

De qualquer forma, a partir de agora, vamos focar apenas nas funções do pacote `stringr`. As funções desse pacote, são em geral, mais rápidas do que as funções ofertadas pelos pacotes básicos. Além disso, os nomes de todas as funções do pacote `stringr` começam pela sequência `str_*`(), o que facilita muito a sua memorização de cada função.

10.5 Comprimento de *strings* com `str_length()`

A função `str_length()` lhe permite contabilizar o número de caracteres presentes em um *string*. Essa função é extremamente útil, quando desejamos aplicar operações que se baseiam em uma determinada posição de um *string*, como extrair uma seção específica desse *string*. Perceba abaixo, que ao se deparar com valores NA, a função nos retorna um valor NA correspondente. Repare também, pelo resultado do quarto elemento, referente a palavra “Partindo”, que espaços em branco também são contabilizados como caracteres, portanto, fique atento a este detalhe.

```
vec <- c(
  "Fui ao Paraná, e encontrei o Varadã",
  "Abril",
  "!",
  "Partindo ",
  NA
)
```

```
str_length(vec)
```

```
## [1] 35  5  1  9 NA
```

10.6 Lidando com capitalização e espaços em branco

Diversas empresas que utilizam formulários, ou outros sistemas de registro, precisam estar constantemente corrigindo *input*'s fornecidos por seus usuários. Talvez, os erros mais comumente gerados, sejam no uso da capitalização e de espaços em branco. Por exemplo, ao preenchermos formulários, é muito comum que: 1) esqueçamos a tecla Caps Lock ligada; 2) ou simplesmente ignoramos o uso de capitalização por simplesmente estarmos com pressa para finalizar o formulário; 3) acrescentar espaços desnecessários ao final ou no meio do *input*.

Como exemplo, suponha que você possua a tabela `usuarios`. Repare que os valores da coluna `cidade`, variam bastante quanto ao uso da capitalização. Repare também, que em alguns valores na coluna `nome`, temos para além de problemas com a capitalização, espaços em branco desnecessários, que as vezes se encontram a direita, ou a esquerda, ou em ambos os lados do nome.

```
usuarios <- tibble(
  nome = c("Ana", " Eduardo", " Cláudio  ", "VerÔNICA ",
           " hugo  ", "JULIANA", " Vitor de paula  "),
  cidade = c("BELÉM", "goiânia", "são paulo", "São paulo", "SÃO PAULO",
            "rIO DE janeiro", "rio de janeiro"),
  profissao = c("Bióloga", "Biólogo", "Químico", "Socióloga",
               "Administrador", "Administradora", "Economista")
)

usuarios
```

```
## # A tibble: 7 x 3
##   nome          cidade      profissao
##   <chr>         <chr>      <chr>
## 1 "Ana"         BELÉM      Bióloga
## 2 " Eduardo"    goiânia    Biólogo
## 3 " Cláudio  "  são paulo  Químico
## 4 "VerÔNICA "  São paulo  Socióloga
## 5 " hugo  "    SÃO PAULO  Administrador
## 6 "JULIANA"    rIO DE janeiro Administradora
## 7 " Vitor de paula  " rio de janeiro Economista
```

No Excel, você normalmente utilizaria a função `ARRUMAR()` para resolver os excessos de espaços, e as funções `MAIÚSCULA()`, `MINÚSCULA()` e `PRI.MAIÚSCULA()` para alterar a capitalização de todas as letras de cada nome. Sendo as funções `str_trim()`, `str_to_upper()`, `str_to_lower()` e `str_to_title()`, os seus equivalentes no pacote `stringr`, respectivamente.

Como os próprios nomes das funções `str_to_upper()` e `str_to_lower()` dão a entender, elas convertem todas as letras contidas em um vetor do tipo `character`, para letras maiúsculas (*upper*) e minúsculas (*lower*). Por exemplo, ao aplicarmos essas funções sobre a coluna `cidade`, temos o seguinte resultado:


```
usuarios %>%
  mutate(cidade = str_to_upper(cidade))

## # A tibble: 7 x 3
##   nome          cidade      profissao
##   <chr>         <chr>      <chr>
## 1 "Ana"         BELÉM      Bióloga
## 2 " Eduardo"   GOIÂNIA    Biólogo
## 3 " Cláudio   " SÃO PAULO  Químico
## 4 "VerÔNICA "  SÃO PAULO  Socióloga
## 5 " hugo     " SÃO PAULO  Administrador
## 6 "JULIANA"    RIO DE JANEIRO Administradora
## 7 " Vitor de paula " RIO DE JANEIRO Economista
```

```
usuarios %>%
  mutate(cidade = str_to_lower(cidade))

## # A tibble: 7 x 3
##   nome          cidade      profissao
##   <chr>         <chr>      <chr>
## 1 "Ana"         belém      Bióloga
## 2 " Eduardo"   goiânia    Biólogo
## 3 " Cláudio   " são paulo  Químico
## 4 "VerÔNICA "  são paulo  Socióloga
## 5 " hugo     " são paulo  Administrador
## 6 "JULIANA"    rio de janeiro Administradora
## 7 " Vitor de paula " rio de janeiro Economista
```

Por outro lado, a função `str_to_title()` representa a alternativa do meio, ao converter a primeira letra de cada palavra, para maiúsculo, e as letras restantes, para minúsculo, como demonstrado abaixo:

```
usuarios %>%
  mutate(cidade = str_to_title(cidade))

## # A tibble: 7 x 3
##   nome          cidade      profissao
##   <chr>         <chr>      <chr>
## 1 "Ana"         Belém      Bióloga
## 2 " Eduardo"   Goiânia    Biólogo
## 3 " Cláudio   " São Paulo  Químico
## 4 "VerÔNICA "  São Paulo  Socióloga
## 5 " hugo     " São Paulo  Administrador
## 6 "JULIANA"    Rio De Janeiro Administradora
## 7 " Vitor de paula " Rio De Janeiro Economista
```

Quanto ao excedente de espaços na coluna *nome*, podemos aplicar a função `str_trim()`. Por padrão, essa função retira qualquer espaço remanescente em ambos os lados de seu *string*. Mas caso seja de seu desejo, você pode especificar um lado específico para retirar espaços, por meio do argumento *side*, que aceita os valores "both", "left" ou "right".

```
usuarios <- usuarios %>%
  mutate(nome = str_trim(nome))

usuarios

## # A tibble: 7 x 3
##   nome      cidade      profissao
##   <chr>      <chr>      <chr>
## 1 Ana        BELÉM        Bióloga
## 2 Eduardo    goiânia      Biólogo
## 3 Cláudio    são paulo    Químico
## 4 VerÔNICA   São paulo    Socióloga
## 5 hugo       SÃO PAULO    Administrador
## 6 JULIANA    rIO DE janeiro Administradora
## 7 Vitor de paula rio de janeiro Economista
```

Vale destacar também, que `str_trim()` é capaz apenas de remover excessos de espaços que se encontram ao redor de seu texto. Logo, a forma mais direta de resolvermos esse tipo de excesso, seria utilizarmos o método mais “abrangente” da função `str_trim()`, aplicado pela função `str_squish()`. Além de remover os espaços ao redor da palavra, a função `str_squish()` também é capaz de remover espaços repetidos que se encontram entre palavras. Veja abaixo, o exemplo do texto " São Carlos de Santana ".

```
str_trim(" São Carlos de Santana ")

## [1] "São Carlos de Santana"

str_squish(" São Carlos de Santana ")

## [1] "São Carlos de Santana"
```

10.7 Extrair partes ou *subsets* de um *string* com str_sub()

Para extrairmos partes de um *string*, podemos utilizar a função `str_sub()`, que se baseia na posição dos caracteres que delimitam o intervalo que você deseja capturar. Ou seja, nessa função, precisamos definir as posições dos caracteres que iniciam e terminam o intervalo que estamos extraindo. Como exemplo, eu posso extrair do primeiro ao quarto caractere de cada texto presente na coluna *nome*, da seguinte maneira:

```
usuarios %>%
  mutate(parte = str_sub(nome, start = 1, end = 4))
```

```
## # A tibble: 7 x 4
```

	nome	cidade	profissao	parte
##	<chr>	<chr>	<chr>	<chr>
## 1	Ana	BELÉM	Bióloga	Ana
## 2	Eduardo	goiânia	Biólogo	Edua
## 3	Cláudio	são paulo	Químico	Cláu
## 4	VerÔNICA	São paulo	Socióloga	VerÔ
## 5	hugo	SÃO PAULO	Administrador	hugo
## 6	JULIANA	rio DE janeiro	Administradora	JULI
## 7	Vitor de paula	rio de janeiro	Economista	Vito

De forma semelhante, podemos extrair do terceiro ao quinto caractere dessa mesma coluna, de acordo com o seguinte comando:

```
usuarios %>%
  mutate(parte = str_sub(nome, start = 3, end = 5))
```

```
## # A tibble: 7 x 4
```

	nome	cidade	profissao	parte
##	<chr>	<chr>	<chr>	<chr>
## 1	Ana	BELÉM	Bióloga	a
## 2	Eduardo	goiânia	Biólogo	uar
## 3	Cláudio	são paulo	Químico	áud
## 4	VerÔNICA	São paulo	Socióloga	rÔN
## 5	hugo	SÃO PAULO	Administrador	go
## 6	JULIANA	rio DE janeiro	Administradora	LIA
## 7	Vitor de paula	rio de janeiro	Economista	tor

Além desses pontos, vale esclarecer que os textos incluídos em seu vetor, não precisam obrigatoriamente se encaixar no intervalo de caracteres que você delimitou. Por exemplo, veja abaixo que eu forneci um vetor contendo dois nomes (Ana e Eduardo), um possui 3 caracteres, enquanto o outro, possui 7. Logo, ao pedir à `str_sub()`, que retire do primeiro ao sexto caractere de cada texto contido no vetor, a função vai tentar extrair o máximo de caracteres possíveis que se encaixam nesse intervalo. Mesmo que algum desses textos não encaixe por completo nesse intervalo.

```
str_sub(c("Ana", "Eduardo"), start = 1, end = 6)
```

```
## [1] "Ana"      "Eduard"
```

10.7.1 Aliando str_sub() com str_length() para extrair partes de tamanho variável

Talvez você se recorde, que nós utilizamos as funções `str_sub()` e `str_length` anteriormente, mais especificamente, na seção **Um estudo de caso: uma demanda real sobre a distribuição de ICMS**. Nessa seção, possuíamos um sistema que coletava o nome de cada planilha que importávamos para o R, e por que precisávamos dessa informação? Porque o nome de cada planilha especificava o mês e o ano a que os seus dados se referiam. Logo, os dados presentes na planilha `Abril_2019.xlsx` diziam respeito ao mês de Abril do ano de 2019, e assim por diante.

Portanto, ao final da coleta desses nomes, inseríamos esses nomes em uma coluna de nosso `data.frame`, tendo como resultado algo parecido com a coluna `origem`, que se encontra na tabela `periodo`, e que pode ser recriada através dos comandos abaixo.

```
meses <- c("Janeiro", "Fevereiro", "Março", "Abril",
           "Maio", "Junho", "Julho", "Agosto",
           "Setembro", "Outubro", "Novembro", "Dezembro")
meses <- rep(meses, times = 6)
anos <- rep(2015:2020, each = 12)

periodo <- tibble(
  origem = str_c(str_c(meses, anos, sep = "_"), ".xlsx")
)
```

`periodo`

```
## # A tibble: 72 x 1
##   origem
##   <chr>
## 1 Janeiro_2015.xlsx
## 2 Fevereiro_2015.xlsx
## 3 Março_2015.xlsx
## 4 Abril_2015.xlsx
## 5 Maio_2015.xlsx
## 6 Junho_2015.xlsx
## 7 Julho_2015.xlsx
## 8 Agosto_2015.xlsx
## 9 Setembro_2015.xlsx
## 10 Outubro_2015.xlsx
## # ... with 62 more rows
```

Com essa informação, podíamos facilmente rastrear a origem de cada linha de nossa tabela. Entretanto, mesmo com essa informação ainda não éramos capazes de ordenarmos a base de maneira útil. Pois da forma como as informações são apresentadas na coluna `origem`, uma ordenação alfabética seria empregada sobre a coluna. Logo, valores como `Abril_2018.xlsx` e `Abril_2017.xlsx`, viriam a aparecer antes de valores como `Março_2019.xlsx`.

Por isso, ainda tínhamos a necessidade de extrair o mês e o ano desses nomes, e em seguida, alocar essas informações em colunas separadas. Com esse objetivo, utilizamos a função `str_sub()` para extrairmos a parte, ou a seção de cada nome, que correspondia ao mês que ele se referia. Porém, como você pode ver acima, o número de caracteres presentes em cada mês, ou em cada nome, varia de maneira drástica.

Em momentos como esse, você pode tentar identificar se a parte final ou a parte inicial dos textos inclusos em sua coluna, são de alguma maneira, constantes. Ou seja, mesmo que o número de caracteres varie muito ao longo da coluna, talvez exista uma parte específica desses textos que sempre possui a mesma **quantidade de caracteres**.

No caso da coluna origem, temos três partes que são sempre constantes, que são a parte dos anos (mesmo que os anos variem, eles sempre são formados por 4 números, ou 4 caracteres), a parte da extensão do arquivo (`.xlsx`), e o *underscore* (`_`), que sempre separa as duas partes anteriores do mês em cada nome. Somando os caracteres dessas três partes, temos sempre 10 caracteres ao final do nome do arquivo, ao qual podemos eliminar para chegarmos a seção do texto que contém o nome do mês. Com isso, podemos utilizar a função `str_length()` para calcular o número total de caracteres de cada texto, e subtrair 10 desse valor, para chegarmos ao caractere que delimita o fim do mês em cada texto.

Podemos empregar a mesma linha de raciocínio, para chegarmos aos limites do intervalo que contém o ano em cada texto. Contudo, tanto o limite inicial quanto o limite final desse intervalo, variam. Logo, teremos de utilizar o resultado de `str_length()` para descobrirmos os dois limites dessa seção. Como estamos empregando os valores produzidos por `str_length()` em três locais diferentes, eu guardo o resultado dessa função em uma coluna denominada `num`, para não ter o trabalho de digitar repetidamente a função `str_length()`.

```
periodo %>%
  mutate(
    num = str_length(origem),
    mes = str_sub(origem, start = 1, end = num - 10),
    ano = str_sub(origem, start = num - 8, end = num - 5) %>% as.integer()
  )
```

```
## # A tibble: 72 x 4
##   origem          num mes      ano
##   <chr>          <int> <chr>  <int>
## 1 Janeiro_2015.xlsx      17 Janeiro  2015
## 2 Fevereiro_2015.xlsx   19 Fevereiro 2015
## 3 Março_2015.xlsx       15 Março    2015
## 4 Abril_2015.xlsx       15 Abril    2015
## 5 Maio_2015.xlsx        14 Maio     2015
## 6 Junho_2015.xlsx       15 Junho    2015
## 7 Julho_2015.xlsx       15 Julho    2015
## 8 Agosto_2015.xlsx      16 Agosto   2015
```

```
## 9 Setembro_2015.xlsx      18 Setembro    2015
## 10 Outubro_2015.xlsx      17 Outubro     2015
## # ... with 62 more rows
```

10.8 Expressões regulares (ou *regex*) com `str_detect()`

Expressões regulares (*regular expressions*), ou simplesmente *regex*, são uma ferramenta extremamente poderosa para processamento de texto. Por essa característica, praticamente toda linguagem de programação possui em algum nível, uma implementação dessa funcionalidade. Você talvez não saiba ainda, mas expressões regulares estão em todo lugar. Como exemplo, quando você pesquisa por uma palavra em um PDF, você está aplicando uma expressão regular ao longo do arquivo.

Em síntese, expressões regulares são como uma mini-linguagem que lhe permite descrever de maneira concisa, um pedaço de texto (FRIEDL, 2006). Para utilizar uma expressão regular, você precisa utilizar uma função que possa aplicar esse tipo de mecanismo. Nos pacotes básicos do R, essa funcionalidade está disponível através das funções da família `grep()` (sendo `grep()`, `grep1()` e `gsub()`), as principais funções dessa família).

Por outro lado, o pacote `stringr` oferece uma família um pouco maior de funções que são capazes de aplicar tal mecanismo. Sendo as funções `str_which()`, `str_detect()`, `str_replace()` e `str_split()`, as principais representantes dessa família.

Em grande parte desse capítulo, estaremos utilizando a função `str_detect()` como a nossa ponte de acesso ao mundo das expressões regulares. Assim como todas as funções `str_*`() que citamos no parágrafo anterior, a função `str_detect()` aceita um vetor contendo os textos a serem pesquisados como primeiro argumento (`string`), e uma expressão regular como seu segundo argumento (`pattern`).

A função `str_which()` é praticamente idêntica à `str_detect()`. Pois ambas as funções vão pesquisar pelos textos que são descritos pela expressão regular que você forneceu, e ambas as funções vão gerar um vetor contendo índices, que definem quais foram os textos encontrados. Entretanto, as funções se divergem no tipo de resultado gerado. A função `str_which()` nos retorna um vetor contendo índices numéricos. Em contrapartida, a função `str_detect()` gera um vetor de valores lógicos. Portanto, você pode utilizar o resultado de ambas as funções dentro da função de *subsetting* (`[]`) para filtrar os textos encontrados, sendo a única diferença, o tipo de índice empregado no filtro.

10.8.1 A expressão regular mais simples de todas

A maneira mais simples de utilizarmos uma expressão regular, seria pesquisarmos por uma sequência específica de letras. Por exemplo, suponha que eu possua o conjunto de palavras presentes em `vec`, e desejasse encontrar a palavra “emissão”.

```
vec <- c("permissão", "demissão", "emissão", "penitência",
        "jurisdição", "ordenação", "concluído", "vantagem",
        "natação", "satisfação", "conclusão", "ilusão")
```

Com o conhecimento que você já possui, você provavelmente tentaria algo como o comando abaixo para encontrar essa palavra.

```
vec[vec == "emissão"]
```

```
## [1] "emissão"
```

Porém, você também poderia encontrar essa palavra inclusa no vetor `vec`, ao fornecer uma expressão regular que seja capaz de descrever o texto “emissão”. Em seu primeiro instinto, você provavelmente aplicaria o simples texto “emissão”, todavia, como vemos abaixo, esse não é exatamente o resultado que desejamos.

```
teste <- str_detect(vec, "emissão")
vec[teste]
```

```
## [1] "demissão" "emissão"
```

O erro acima, está no fato de que estamos interpretando a **expressão regular** “emissão”, como a palavra “emissão”. Você rapidamente irá descobrir, que expressões regulares não possuem qualquer noção do que é uma palavra, muito menos de onde uma começa ou termina. Ou seja, quando estiver utilizando expressões regulares, a menos que você defina explicitamente os limites físicos da pesquisa, o mecanismo estará procurando por uma sequência específica de caracteres, independente do local em que essa sequência seja detectada.

Por isso, é importante que você comece a interpretar qualquer expressão regular, como uma descrição de uma sequência específica de caracteres, ao invés de uma palavra. Logo, quando fornecemos o texto “emissão” à `str_detect()` acima, estávamos na verdade, buscando qualquer texto que contesse os caracteres “e-m-i-s-s-ã-o”, precisamente nessa ordem. Com isso, a palavra “demissão” foi incluída no resultado acima, pelo fato de possuir tal sequência de caracteres, mesmo que essa sequência esteja acompanhada por um “d”, o qual não faz parte da expressão regular definida.

Como um outro exemplo, suponha que eu utilize a expressão “is”. Lembre-se que nós não estamos procurando pela palavra *is*, mas sim, por qualquer texto que contenha um “i” imediatamente seguido por um “s”. Marcando de negrito, apenas as partes dos textos abaixo, que foram de fato encontradas pela expressão “is”, temos: **satis**fação, **demis**sação, **permiss**sação, **emiss**sação, **juris**dição.

```
teste <- str_detect(vec, "is")
vec[teste]
```

```
## [1] "permissão" "demissão" "emissão" "jurisdição" "satisfação"
```

Porém, a partir do momento em que acrescento um segundo “s” à expressão, as palavras “jurisdição” e “satisfação” não mais se encaixam na descrição fornecida pela expressão. Pois nenhuma dessas duas palavras possuem, **em algum lugar**, um “i” imediatamente seguido por duas letras “s”. Com isso, temos que as partes localizadas pela expressão são: **permiss**sação, **demis**sação, **emiss**sação.

```
teste <- str_detect(vec, "iss")
vec[teste]

## [1] "permissão" "demissão" "emissão"
```

Apenas para que os pontos abordados fiquem claros, a figura exposta abaixo, lhe permite visualizar as correspondências (marcadas em cinza) encontradas por cada uma das expressões regulares mostradas anteriormente.

Figura 10.2: Correspondências encontradas por cada expressão regular, além de suas respectivas descrições.

emissão demissão Localiza:	• Expressão regular: emissão • Descrição: encontra textos que possuem, em algum lugar, a sequência "e-m-i-s-s-ã-o" de caracteres.
permissão demissão emissão jurisdição satisfação Localiza:	• Expressão regular: is • Descrição: encontra textos que possuem, em algum lugar, a sequência "i-s" de caracteres. Ou em outras palavras, textos que possuem uma letra "i" imediatamente seguida de uma letra "s".
permissão demissão emissão Localiza:	• Expressão regular: iss • Descrição: encontra textos que possuem, em algum lugar, a sequência "i-s-s" de caracteres. Ou em outras palavras, textos que possuem uma letra "i" imediatamente seguida por duas letras "s".

Fonte: Elaboração própria do autor.

10.8.2 Caracteres literais e *metacharacters*

Expressões regulares são uma linguagem formada por duas categorias de caracteres (FRIEDL, 2006): 1) Caracteres literais, ou simples letras e números pelos quais pesquisamos; e 2) *metacharacters*, que são um conjunto de caracteres especiais que delimitam o escopo de sua pesquisa, ou a maneira como ela será executada.

Até o momento, utilizamos apenas caracteres literais, ao pesquisarmos pelas sequências "emissão" ou "is". Ou seja, qualquer número ou letra que formam uma sequência de caracteres são considerados caracteres literais. Alguns símbolos também são considerados caracteres literais, pois não possuem nenhum comportamento especial que altere o comportamento da pesquisa. Como exemplo, a expressão "A1_z-4!D8" é formada apenas por caracteres literais, mesmo que ela descreva uma sequência bem esquisita (e provavelmente inútil) de caracteres.

Qualquer expressão que utilize apenas caracteres literais, busca efetuar uma simples pesquisa por uma sequência particular de caracteres. Consequentemente, a expressão "1" é capaz de detectar o texto "Álvaro chegou em 1º lugar!", assim como "O aluguel chegou a R\$3250,10 nesse mês". Como um outro exemplo, ao empregarmos a expressão "regi", ela é capaz de encontrar os textos "região" e "registro", mas não é capaz de detectar o nome "Reginaldo", pelo simples fato de que a primeira letra do nome é um "r" maiúsculo, ao invés de um "r" minúsculo.

Em síntese, expressões regulares já são uma ferramenta útil apenas com o uso de caracteres literais. Contudo, elas se tornam bastante limitadas sem o uso de *metacharacters*, que ampliam em muito as suas funcionalidades, e mudam drasticamente a forma como a pesquisa ocorre. Neste ponto, também reside uma importante dificuldade no domínio de expressões regulares. Pois são muitos *metacharacters* disponíveis e, por isso, memorizar o que cada um deles fazem, e quais são as suas aplicações mais úteis, não se trata de uma tarefa simples.

Apesar disso, haverá momentos em que você deseja encontrar ou incluir em sua expressão regular o caractere literal que um certo *metacharacter* representa. Em outras palavras, há ocasiões em que você deseja que certos *metacharacters* se comportem como caracteres literais. Por exemplo, um dos *metacharacters* que vamos mostrar nas próximas seções é ? (ponto de interrogação). Portanto, o caractere ? possui um comportamento especial em expressões regulares, mas se quisermos encontrar o caractere ? em si, ao longo do texto, nós precisamos contornar o comportamento especial desse *metacharacter*. Para isso, basta anteceder esse caractere por uma barra inclinada à esquerda (\?).

Porém, lembre-se que para escrevermos uma barra inclinada à esquerda, nós temos que digitar duas barras inclinadas à esquerda! Logo, para escrever em sua expressão regular, o termo \?, você deve na verdade, digitar o termo \\?. Isso funciona para praticamente qualquer *metacharacter*. Logo, sempre que você precisar utilizar um certo *metacharacter* como um caractere literal, tente antecederlo por duas barras inclinadas à esquerda.

10.8.3 Âncoras (*anchors*)

O primeiro tipo de *metacharacters* que vou apresentar, são os do tipo "âncora". Esse conjunto é composto pelos caracteres ^ e \$, que são responsáveis por delimitar o início e o fim de uma linha, respectivamente.

Logo, ao utilizar a expressão "^emissão\$", eu estou pedindo à `str_detect()` que localize um texto que contém: o início de uma linha imediatamente seguido pela sequência "e-m-i-s-s-ã-o" de caracteres, que por sua vez, deve ser imediatamente seguido pelo fim dessa mesma linha. Com

essa expressão, somos capazes de encontrar apenas a palavra “emissão” que está entre os valores do vetor `vec`.

```
teste <- str_detect(vec, "^emissão$")
vec[teste]
```

```
## [1] "emissão"
```

É importante destacar, que os caracteres `^` e `$` são capazes de encontrar os limites de uma linha, e não de uma palavra. Por isso, a partir do momento em que a sequência “e-m-i-s-s-ã-o” não estiver encostando em pelo menos um dos limites da linha, `str_detect()` não será mais capaz de encontrar tal conjunto de caracteres. Como exemplo, perceba abaixo, que apenas o primeiro elemento de `text` pôde corresponder à expressão empregada em `str_detect()`. Ou seja, mesmo que o quarto, quinto e sexto elementos de `text` possuam a palavra “emissão”, eles não puderam ser encontrados pela expressão `^emissão$`, devido ao fato de não estarem localizados em pelo menos um dos limites da linha.

```
text <- c(
  "emissão",
  "A Ford Brasil executou recentemente uma demissão em massa",
  "remissão",
  "Para mais, a emissão de CO2 cresceu no Brasil",
  "emissão de SO2 faz parte do processo",
  "A firma foi processada por tal emissão"
)
```

```
teste <- str_detect(text, "^emissão$")
text[teste]
```

```
## [1] "emissão"
```

Vale destacar que você não precisa necessariamente utilizar os dois *metacharacters* ao mesmo tempo. Logo, temos a opção de construir uma expressão que possa encontrar uma certa sequência de caracteres ao final ou no início de uma linha. Por exemplo, a expressão abaixo, busca encontrar a sequência “e-m-i-s-s-ã-o” de caracteres quando ela é imediatamente seguida pelo final da linha.

```
teste <- str_detect(text, "emissão$")
text[teste]
```

```
## [1] "emissão"
## [2] "remissão"
## [3] "A firma foi processada por tal emissão"
```

Alguns outros exemplos de expressões regulares que empregam *metacharacters* do tipo âncora, além de uma rápida reflexão sobre os caracteres `^` e `$`, são oferecidos na figura abaixo. Repare que todas as partes do texto que foram detectadas pela expressão regular, foram novamente marcadas de cinza. Perceba também, que cada seta presente na figura, busca conectar cada uma das partes detectadas do texto, ao componente específico da expressão regular que foi responsável por detectá-la.

10.8.4 Classes de caracteres (*character classes*)

Uma estrutura muito importante em expressões regulares são as classes de caracteres, ou *character classes*. Sendo construída a partir de um par de colchetes (`[]`), essa estrutura lhe permite listar os possíveis caracteres que você deseja encontrar em um ponto da sequência descrita por sua expressão regular.

Por exemplo, suponha que você esteja lendo um livro-texto sobre a linguagem R, e que você queira encontrar todas as instâncias do livro que se referem ao termo *regex*. Você sabe que as regiões que descrevem o assunto no qual você está interessado, vão conter o termo *regex*, mas você não sabe como o termo *regex* está citado no texto. Digo, será que o autor está colocando a primeira letra em maiúsculo (Regex)? Ou será que todo o termo está em maiúsculo (REGEX)?

Tendo essa dúvida em mente, você pode criar uma expressão regular, que permita certas variações da palavra *regex*, ao listar todas as possibilidades em uma dada posição do termo. Primeiro, vamos imaginar que você deseja permitir que a primeira letra do termo seja tanto maiúscula quanto minúscula. No exemplo abaixo, ao incluirmos as letras “r” e “R” dentro da classe de caracteres (`[]`), estamos estabelecendo que no primeiro caractere da sequência, podemos ter uma letra “r” ou uma letra “R”.

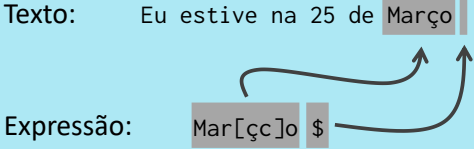
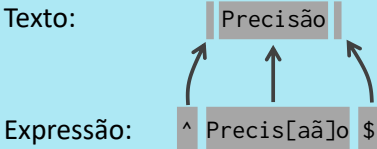
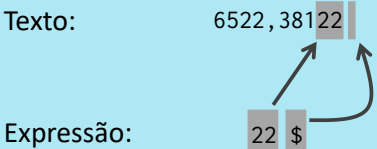
```
texto <- c(
  "Cada letra, número, ou símbolo presente no texto é um caractere.",
  "Textos são criados ao contornados por aspas (duplas ou simples).",
  "O termo regex é uma abreviação para regular expressions.",
  "Regex é um termo comum no mundo da computação.",
  "Metacharacters alteram consideravelmente o comportamento de um REGEX.",
  "ReGEx? Ou reGex? Talvez RegEX?."
)
```

```
teste <- str_detect(texto, "[Rr]egex")
texto[teste]
```

```
## [1] "O termo regex é uma abreviação para regular expressions."
## [2] "Regex é um termo comum no mundo da computação."
```

Ou seja, uma classe de caracteres busca descrever os caracteres possíveis para uma única e particular posição da sequência. Logo, a expressão `"[Rr]egex"` não está descrevendo a sequência `"[-R-r]-e-g-e-x"`, mas está afirmando que `"r-e-g-e-x"` e `"R-e-g-e-x"` são duas sequências de caracteres que

Figura 10.3: Exemplos e uma reflexão sobre as correspondências encontradas por metacharacters do tipo âncora.

Texto: Eu estive na 25 de Março Expressão: Mar[çc]o \$ 	• Expressão regular: Mar[çc]o\$ • Descrição: encontra textos que possuem a sequência "M-a-r-ç-o" ou "M-a-r-c-o" ao final de uma linha.
Texto: Precisão Expressão: ^ Preci[aã]o \$ 	• Expressão regular: ^Preci[aã]o\$ • Descrição: encontra textos que possuem o início da linha, imediatamente seguido pela sequência "P-r-e-c-i-s-ã-o" ou "P-r-e-c-i-s-a-o", que por sua vez é imediatamente seguido pelo final da linha.
Texto: 6522,38122 Expressão: 22 \$ 	• Expressão regular: 22\$ • Descrição: encontra textos que possuem a sequência "2-2" ao final de uma linha.
Reflexão: • O que as expressões regulares "^\$" e "^ \$" significam? Muitas pessoas tendem a crer que essa é uma pergunta enganosa e, por isso, muitos acabam procurando por diferentes interpretações dos <i>metacharacters</i> ^ e \$. Mas na realidade, essa é uma questão bem simples. Se os <i>metacharacters</i> ^ e \$, representam o início e o fim de uma linha, respectivamente, podemos dizer que a expressão "^\$" busca encontrar uma linha que esteja completamente vazia. Por outro lado, como há um espaço em branco separando os <i>metacharacters</i> ^ e \$ na expressão "^ \$", essa expressão busca encontrar qualquer linha que esteja preenchida com um único espaço, e nada mais.	

Fonte: Elaboração própria do autor.

queremos encontrar em nossa pesquisa. Com isso, se tivéssemos de permitir todas as possibilidades de capitalização em cada letra do termo, poderíamos fornecer a seguinte expressão à `str_detect()`:

```
teste <- str_detect(texto, "[Rr][Ee][Gg][Ee][Xx]")
texto[teste]

## [1] "O termo regex é uma abreviação para regular expressions."
## [2] "Regex é um termo comum no mundo da computação."
## [3] "Metacharacters alteram consideravelmente o comportamento de um REGEX."
## [4] "ReGEx? Ou reGex? Talvez RegEX?."
```

Dessa maneira, estamos permitindo que `str_detect()` encontre todas as possibilidades do termo *regex*, quanto ao uso de capitalização (*regex*, *Regex*, *REGEX*, *rEgex*, *reGex*, *regEx*, *regeX*, ...).

As classes de caracteres também são muito utilizadas para criar um intervalo de caracteres possíveis em um determinado ponto. Esses intervalos são rapidamente formados pelo *metacharacter* - (sinal de menos). Como exemplo, podemos utilizar o atalho `[0-9]` para listarmos todos os números de 0 a 9 dentro da classe. Esse atalho é extremamente útil quando desejamos encontrar alguma parte numérica em nosso texto, mas nós não sabemos previamente quais números em particular vão estar presentes nesse item.

Por exemplo, suponha que uma comissão nacional tenha divulgado as colocações de diversos participantes em um torneio de xadrex. Você deseja analisar os participantes e suas respectivas colocações, entretanto, a comissão divulgou os dados como um texto simples em sua página da internet, ao invés de guardar esses dados em uma tabela, ou em alguma outra estrutura que fosse de fácil transposição para o R.

Com isso, você precisa utilizar uma expressão regular que possa encontrar essas colocações ao longo do texto. Uma possibilidade, seria tentarmos localizar as ocorrências de um número seguido do símbolo de grau (°), ao longo do texto. No exemplo abaixo, as colocações variam de 1 a 6 e, por isso, precisamos listar todos os números neste intervalo dentro de uma classe, e acrescentar o símbolo de grau, formando assim, a expressão `"[123456]°"`. Porém, ao invés de listarmos número por número, podemos aplicar o atalho `[1-6]` para criarmos uma lista contendo todos os números de 1 a 6.

```
colocacoes <- c(
  "1°: Álvaro",
  "2°: Melissa",
  "3°: Ana",
  "4°: Eduardo",
  "5°: Daniela",
  "6°: Matheus",
  "Não é uma colocação",
  "Também não é uma colocação",
  "31°C",
```

```

"24°F"
)

teste <- str_detect(colocacoes, "[1-6]°")
colocacoes[teste]

## [1] "1°: Álvaro" "2°: Melissa" "3°: Ana"      "4°: Eduardo" "5°: Daniela"
## [6] "6°: Matheus" "31°C"          "24°F"
```

Como podemos ver acima, conseguimos localizar todas as colocações. No entanto, perceba que a expressão "[1-6]°" também pôde encontrar informações que se referem a temperaturas (celsius e fahrenheit). Portanto, a expressão "[1-6]°" é muito abrangente para o nosso caso e, em função disso, precisamos descrever em mais detalhes o texto que desejamos. Tudo o que precisamos fazer para corrigir o resultado acima, é incluir uma expressão que encontre um número seguido por um símbolo de grau, **exceto** quando as letras C ou F estão logo após o símbolo de grau.

Para essa tarefa, podemos utilizar o comportamento negativo de uma classe. Em outras palavras, além de listar os caracteres aceitos em uma certa posição, nós também temos a capacidade de utilizar uma classe de caracteres para listar todos os caracteres que **não podem** estar situados em uma determinada posição da sequência.

Para definir os caracteres não desejados em uma posição, você deve iniciar a sua classe, por um acento circunflexo, logo antes de listar os caracteres em questão ([^...]). Com isso, se desejamos evitar as letras C e F (independente de sua capitalização) precisaríamos da sub-expressão [^CcFf] logo após o símbolo de grau, formando assim, a expressão regular abaixo:

```

teste <- str_detect(colocacoes, "[1-6]°[^CcFf]")
colocacoes[teste]

## [1] "1°: Álvaro" "2°: Melissa" "3°: Ana"      "4°: Eduardo" "5°: Daniela"
## [6] "6°: Matheus"
```

Portanto, sempre que você encontrar uma classe que contém um acento circunflexo como seu primeiro item, você sabe que essa classe está negando os caracteres listados dentro dela (exemplo: "[^1-6_!]", não são permitidos nessa posição qualquer número entre 1 e 6, o símbolo *underscore* ou um ponto de exclamação). Logo, na posição que essa classe representa, não devem ser encontrados os caracteres que estão listados dentro dela. Mas se essa classe não possui tal acento, ou se esse acento circunflexo se encontra a partir do segundo caractere listado, a classe em análise está utilizando seu comportamento positivo (ou afirmativo), de modo que os caracteres listados em seu interior, podem sim estar naquela posição.

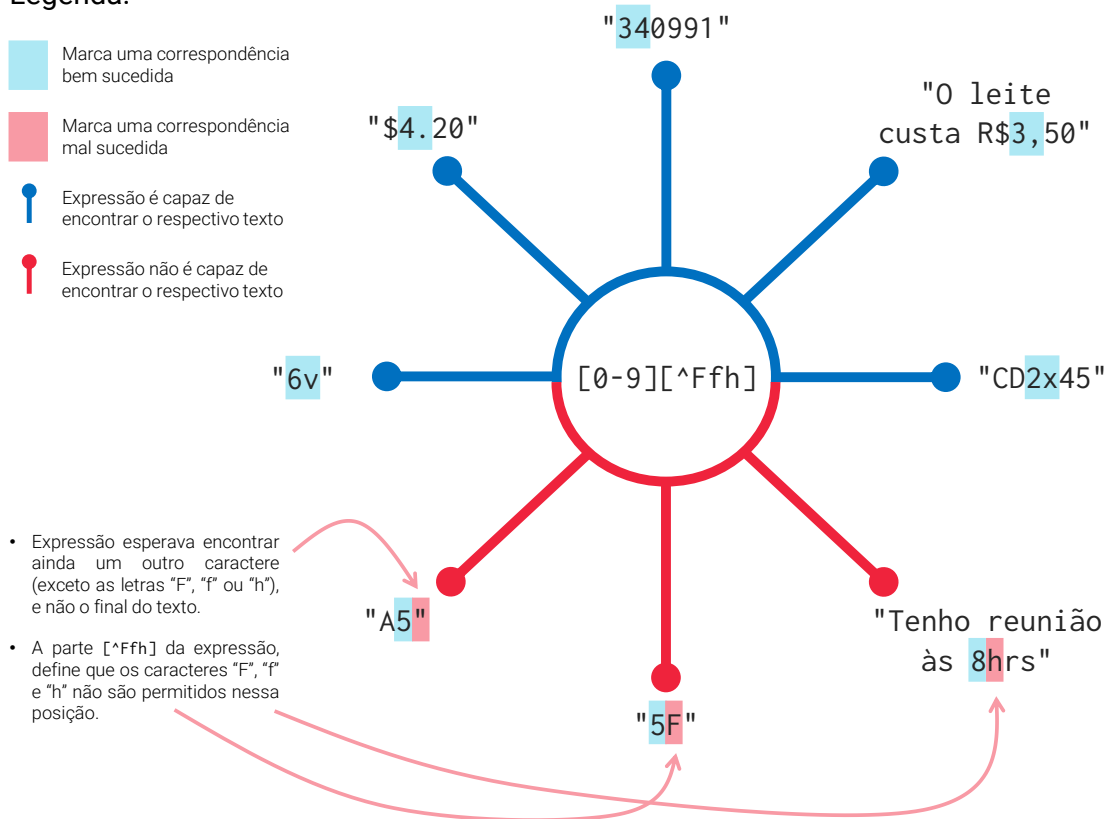
Como um outro exemplo, veja abaixo, as correspondências geradas pela expressão "[0-9][^Ffh]", que utiliza ambos os modos de classe (negativa e positiva). Essa expressão, busca encontrar um número entre 0 e 9, que é imediatamente seguido por um caractere qualquer (que não seja as letras "F", "f" e "h"). Repare no caso do texto "A5", no qual a expressão não é capaz de localizá-lo pelo

simples fato de que o texto acaba no dígito 5. Lembre-se que cada classe de caracteres representa um caractere a ser encontrado em uma determinada posição da sequência. Logo, mesmo que a parte `[^Ffh]` esteja listando os caracteres que não podem ser encontrados, ela está automaticamente definindo que **algum caractere deve ser encontrado na segunda posição da sequência**.

Figura 10.4: Um exemplo de expressão regular que emprega ambos os modos de classes de caractere (positiva e negativa)

Legenda:

- Marca uma correspondência bem sucedida
- Marca uma correspondência mal sucedida
- Expressão é capaz de encontrar o respectivo texto
- Expressão não é capaz de encontrar o respectivo texto



Fonte: Elaboração própria do autor.

Além desses pontos, repare acima, que o *metacharacter* `^` (acento circunflexo) tem um papel completamente diferente dentro de uma classe de caracteres, se comparado ao papel que ele exerce fora dela. Em resumo, o caractere `^` fora de uma classe, é um *metacharacter* do tipo âncora, sendo capaz de definir o início de uma linha; mas dentro de uma classe, ele está determinando o comportamento adotado pela classe em questão, de forma que os caracteres listados nessa classe não devem ser encontrados na posição que essa classe simboliza.

Logo, é muito importante destacar o fato de que diversos caracteres possuem um comportamento profundamente diferente, quando inseridos em uma classe de caracteres. Fique atento a isso! Se

algun *metacharacter* estiver se comportando de maneira inesperada, é possível que essa diferença entre os mundos de dentro e de fora de uma classe seja a fonte de sua surpresa. De certo modo, você pode compreender essa situação, como se as classes possuíssem a sua própria mini-linguagem, com o seu próprio conjunto de *metacharacters*, separados da realidade de fora delas (FRIEDL, 2006).

Por outro lado, e se você desejasse incluir os *metacharacters* - e ^ como possíveis caracteres para uma determinada posição? Como o caractere - cria uma sequência, basta que você liste ele logo no início de sua classe (ex: "[-1-6]", que permite um número entre 1 e 6, além de um sinal de menos). Já o caractere ^, precisa ser posicionado como primeiro item da classe para exercer o seu comportamento especial e, por essa razão, você precisa apenas listá-lo em uma outra posição para se comportar como um simples acento circunflexo (ex: "[ABC^]", que permite as letras A, B e C, além de um acento circunflexo).

```
a <- c("A-B", "CDE-F", "12^54", "R$1230,2", "BRA")
teste <- str_detect(a, "[-^]")
a[teste]

## [1] "A-B" "CDE-F" "12^54"
```

Até o momento, mostramos apenas o atalho para listar uma sequência numérica (ex: "[0-9]"). Mas também temos um outro atalho para listarmos um intervalo específico (ou todas as letras) do alfabeto. Para isso, podemos utilizar o atalho [a-z] para letras minúsculas, e [A-Z] para letras maiúsculas. Por exemplo, suponha que você possua o conjunto de códigos mostrados no objeto codes. Suponha também, que os códigos que contém letras de “A” a “F”, correspondem a unidades manufaturadas em Belo Horizonte, enquanto os códigos que contém letras de “G” a “Z” dizem respeito a unidades fabricadas na região de São Paulo.

Com isso em mente, para reunirmos todos os códigos de produtos construídos em Belo Horizonte, precisaríamos apenas encontrar os códigos que contém qualquer letra dentro do intervalo de “A” e “F”. Todavia, repare que a capitalização das letras presentes nos códigos, varia. Por isso, precisamos combinar o mesmo intervalo de letras em ambos os estilos de capitalização. Dessa maneira, geramos a expressão abaixo, que contém ambos os intervalos ("[a-fA-F]").

```
codes <- c("AeF15", "CCd31", "17GHJ", "Lmm96", "ee3f8", "BA45B",
          "EccF2", "675Cc", "hkJ78", "q401Q", "iop67", "DCa98")
teste <- str_detect(codes, "[a-fA-F]")
codes[teste]

## [1] "AeF15" "CCd31" "ee3f8" "BA45B" "EccF2" "675Cc" "DCa98"
```

10.8.4.1 Conclusão e algumas dicas extras

Portanto, uma classe de caracteres busca listar os caracteres que podem ou não ser encontrados na posição da sequência que essa classe representa. Em síntese, podemos interpretar o seu uso da seguinte maneira:

- `[abc]`: encontre a ou b ou c.
- `[^abc]`: encontre qualquer caractere, exceto a, b ou c.

Além disso, uma classe de caracteres lhe permite criar *ranges*, ou intervalos de caracteres possíveis, como:

- `[0-9]`: encontre qualquer número entre 0 e 9.
- `[a-z]`: encontre qualquer letra (minúscula) entre a e z.
- `[A-Z]`: encontre qualquer letra (maiúscula) entre A e Z.

Porém, para além dos usos apresentados até aqui, o R nos oferece alguns atalhos para essas construções, sendo os principais:

- `\d`: encontre um dígito (atalho para `[0-9]`).
- `\s`: encontre qualquer espaço em branco (atalho para `[\t\n]`).
- `\w`: encontre um caractere alfanumérico ou um *underline* (atalho para `[a-zA-Z0-9_]`)

Lembre-se que, no R, para inserirmos uma barra inclinada à esquerda em um *string*, nós precisamos escrever duas barras inclinadas à esquerda. Logo, para inserirmos, por exemplo, o atalho `\d` em alguma de nossas expressões regulares, somos obrigados a digitar `\\d`.

10.8.5 Representando qualquer caractere com um ponto

Você pode representar qualquer caractere em uma expressão regular, por meio do *metacharacter* `.` (ponto final). Ou seja, um ponto final em uma expressão regular é capaz de encontrar qualquer caractere (seja ele um número, um símbolo ou uma letra) na posição que ele representa. Logo, a expressão `"B.3"` significa na prática: uma letra “B”, imediatamente seguida por um caractere qualquer, que por sua vez, é imediatamente seguido por um número 3.

Por exemplo, suponha que você queira encontrar a data “20/02/2019”, mas você sabe que essa data pode se encontrar em diferentes formatos, como 20.02.2019, ou 20-02-2019. Tendo isso em mente, você provavelmente tentaria uma expressão como `"20[-/.]02[-/.]2019"`. Por outro lado, poderíamos atingir o mesmo resultado ao substituírmos as classes de caracteres por pontos finais, gerando assim, a expressão `"20.02.2019"`.

```
vec <- c("20.02.2019", "20-02-2019", "20/02/2019",
        "A senha é 2060212019", "20$02#2019")
```

```
teste <- str_detect(vec, "20.02.2019")
vec[teste]
```

```
## [1] "20.02.2019"          "20-02-2019"          "20/02/2019"
## [4] "A senha é 2060212019" "20$02#2019"
```

Porém, é importante que você tenha cuidado ao utilizar esse *metacharacter*. Pois como podemos ver acima, a expressão "20.02.2019" também é capaz de encontrar o texto "20\$02#2019", assim como o texto "A senha é 2060212019". Portanto, as chances de você encontrar o que você não deseja, podem aumentar a depender da maneira em que você aplica esse *metacharacter* em sua expressão.

10.8.6 Criando alternativas (*alternation*)

Há certos momentos, em que não conseguimos expor todos os nossos desejos com uma única expressão. Por essa razão, temos o *metacharacter* | (barra vertical) que nos permite combinar diferentes sub-expressões em uma só. Dessa maneira, a função responsável pela pesquisa, irá procurar por qualquer texto que atenda a pelo menos uma dessas sub-expressões. Sendo este efeito, comumente denominado de alternância (ou *alternation*).

Como exemplo, na seção anterior estávamos tentando encontrar o termo *regex*, ao longo de várias sentenças, que estão reproduzidas logo abaixo, no vetor texto. Na primeira instância, fizemos uso de uma classe de caracter para permitirmos uma letra "r" tanto minúscula quanto maiúscula, no primeiro caractere da sequência de nossa expressão ("[Rr]egex").

Porém, temos a capacidade de atingir o mesmo resultado, com o uso de alternância. Basta separarmos os dois casos que estamos tentando representar, pelo *metacharacter* |, formando assim, a expressão abaixo ("Regex|regex"):

```
texto <- c(
  "Cada letra, número, ou símbolo presente no texto é um caractere.",
  "Textos são criados ao contornados por aspas (duplas ou simples).",
  "O termo regex é uma abreviação para regular expressions.",
  "Regex é um termo comum no mundo da computação.",
  "Metacharacters alteram consideravelmente o comportamento de um REGEX.",
  "ReGEx? Ou reGex? Talvez RegEX?."
)

teste <- str_detect(texto, "Regex|regex")
texto[teste]

## [1] "O termo regex é uma abreviação para regular expressions."
## [2] "Regex é um termo comum no mundo da computação."
```

Lembre-se que a realidade dentro de uma classe de caracteres é completamente diferente de seu exterior. Logo, dentro de uma classe de caracteres, o caractere | é simplesmente um caractere literal, assim como as letras "x" e "r". Por isso, uma expressão como "Rege[x|r]egex", estaria na verdade procurando por sequências como "R-e-g-e-x-e-g-e-x", "R-e-g-e-|-e-g-e-x" e "R-e-g-e-r-e-g-e-x".

Para mais, é importante que você entenda que cada sub-expressão conectada pelo *metacharacter* |, representa uma expressão regular diferente das demais.

Veja como exemplo, a expressão abaixo. A primeira sub-expressão ("`[3-6]°`") seleciona um texto que contenha um número entre 3 e 6 imediatamente seguido de um símbolo de grau. A segunda sub-expressão ("`is[ao]`") seleciona um texto que contenha a sequência “i-s-a” ou “i-s-o” de caracteres. Já a terceira sub-expressão ("`R\\$[0-9]+(,[0-9][0-9])?`"), que é bem mais elaborada do que as outras duas, busca selecionar um texto que contenha um valor monetário. Com isso, qualquer texto que se encaixe em alguma dessas condições, será selecionado pela função.

```
vec <- c("1230", "Tenho consulta no dia 25", "R$12,45",
        "Essa máquina custa R$320,21", "Márcia", "Isotônico",
        "Álcool isopropílico", "Hoje fez 30°", "4° é muito frio!")
teste <- str_detect(vec, "[3-6]°|is[ao]|R\\$[0-9]+(,[0-9][0-9])?")
vec[teste]
```

```
## [1] "R$12,45"                "Essa máquina custa R$320,21"
## [3] "Álcool isopropílico"    "4° é muito frio!"
```

Um outro detalhe importante, é que você pode limitar o alcance das alternativas, ao contorná-las com parênteses. Em outras palavras, ao invés de fornecer várias sub-expressões, você pode fornecer diferentes sub-expressões **dentro** de uma expressão “geral”.

Por exemplo, vamos voltar à expressão "`Regex|regex`". Se nós isolarmos a seção "`ex|re`", temos um resultado completamente diferente do que vimos anteriormente, pois as sub-expressões passam a ser “e-x” e “r-e”, e não “r-e-g-e-x” e “R-e-g-e-x” como anteriormente. Dessa maneira, estamos na verdade procurando por textos que contenham a sequência “R-e-g-e-x-g-e-x” ou a sequência “R-e-g-r-e-g-e-x”.

```
vec <- c("regex", "Regex", "ISORegex-18930", "Regexgexgexgexgex")
teste <- str_detect(vec, "Reg(ex|re)gex")
vec[teste]
```

```
## [1] "Regexgexgexgexgex"
```

Dessa vez, importando um exemplo diretamente da obra de [Friedl \(2006\)](#), suponha que você possua um arquivo de texto, contendo uma lista de todos os emails de sua caixa de entrada. Com esse arquivo, poderíamos utilizar a expressão "`^(From|Subject|Date):`" para extraírmos apenas as linhas do arquivo que contém a referência do remetente (From:), do assunto (Subject:) e da data de envio (Date:) de cada email. Perceba também, que a expressão "`^(From|Subject|Date):`" é equivalente à expressão "`^From:|^Subject:|^Date:`".

```
email <- readr::read_lines("
From: elena_campaio@gmail.com Jun 15 2019 07:05
Received: from elena_campaio@gmail.com
To: pedropark99@gmail.com
Date: Thu, Jun 15 2019 07:05
```

Message-Id: <20190322145154232.elena_campaio@gmail.com>
 Subject: Nova reunião
 X-Mailer: by mailbox (Version 8.5.1) BellM Company, Copyright 2005-2019

Bom dia Pedro, poderíamos nos encontrar às 10hrs?

From: pedropark99@gmail.com Jun 15 2019 08:10
 Received: from elena_campaio@gmail.com
 To: elena_campaio@gmail.com
 Date: Thu, Jun 15 2019 08:10
 Message-Id: <20190322145155198.elena_campaio@gmail.com>
 Subject: Re: Nova reunião
 Reply-To: elena_campaio@gmail.com <20190322145154232.elena_campaio@gmail>
 X-Mailer: by mailbox (Version 8.5.1) BellM Company, Copyright 2005-2019

Ok Elena! Podemos nos encontrar esse horário.")

```
teste <- str_detect(email, "^(From|Subject|Date):")
email[teste]
```

```
## [1] "From: elena_campaio@gmail.com Jun 15 2019 07:05"
## [2] "Date: Thu, Jun 15 2019 07:05"
## [3] "Subject: Nova reunião"
## [4] "From: pedropark99@gmail.com Jun 15 2019 08:10 "
## [5] "Date: Thu, Jun 15 2019 08:10"
## [6] "Subject: Re: Nova reunião"
```

10.8.7 Quantificadores (*quantifiers*) ou definindo repetições

Há certos momentos em que precisamos permitir que um certo conjunto de caracteres sejam encontrados múltiplas vezes em uma mesma sequência de caracteres. Um bom exemplo disso, é a expressão que utilizamos na seção anterior "R\\\$[0-9]+(,[0-9][0-9])?" para encontrarmos um valor monetário. Temos três partes principais nessa expressão, sendo elas: 1) R\\\$; 2) [0-9]+; e 3) (,[0-9][0-9])?.

Primeiro, o que seria um valor monetário? Certamente seria um valor numérico. Porém, um número pode significar qualquer coisa! Talvez uma medida de peso (Kg), idade (anos), volume (L) ou qualquer outra variável contínua que você imaginar. Logo, precisamos de algum item que possa identificar esse número como uma medida de valor, e esse item se trata do símbolo da moeda brasileira (R\$). Qualquer valor numérico presente em seu texto que estiver acompanhado desse símbolo é um valor monetário.

Com isso, teríamos a expressão "R\\\$[0-9]" como uma tentativa inicial. Perceba que eu tive de contornar o comportamento especial do *metacharacter* \$, ao anteceder-lo por duas barras inclinadas.

Dessa maneira, a expressão "\\\$" significa de fato o caractere \$ (cifrão), e não o fim de uma linha como definimos na seção *Âncoras (anchors)*.

```
vec <- c("Eu peso em torno de 65Kg", "Tenho consulta no dia 25",
        "R$1630,45", "Eu possuo uma conta de R$74,85 a pagar",
        "R$400", "R21", "Hoje, R$30 equivale a $5,77 dólares")
teste <- str_detect(vec, "R\\$[0-9]")
vec[teste]

## [1] "R$1630,45"
## [2] "Eu possuo uma conta de R$74,85 a pagar"
## [3] "R$400"
## [4] "Hoje, R$30 equivale a $5,77 dólares"
```

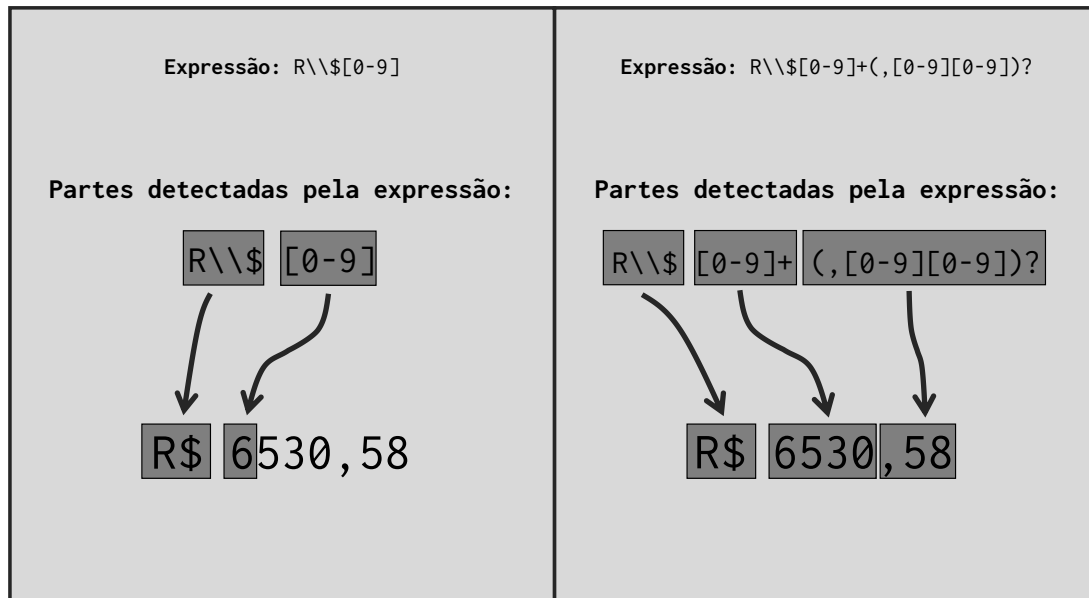
Entretanto, não há um limite específico para o número que um valor monetário pode atingir. Em outras palavras, podemos estar nos referindo a míseros centavos ou a milhões de reais. Traduzindo essa afirmação na prática, podemos ter uma quantidade variável de dígitos em nosso valor monetário. O valor R\$5 possui apenas 1 dígito, enquanto o valor R\$1245 possui 4 dígitos.

A princípio, essa questão não é tão importante, já que fomos capazes de encontrar todos os textos que contém algum valor monetário, com apenas a expressão "R\\\$[0-9]". Ou seja, mesmo que alguns desses valores possuam 3, 4 ou 6 dígitos, precisamos apenas detectar o seu primeiro dígito antecedido pelo símbolo R\$.

Todavia, essa questão passa a ser crucial, na hipótese de aplicarmos alguma transformação sobre os valores monetários encontrados. Ou seja, se vamos, por exemplo, extrair os valores encontrados; ou substituí-los por algum outro texto; ou utilizá-los como pontos de quebra do texto que os contém; ou empregá-los em algum cálculo, é de extrema importância que possamos detectar todo o valor com a nossa expressão. Apenas para que fique claro, veja a representação abaixo, que mostra os resultados de ambas as expressões mostradas até aqui sobre o valor R\$6530,58.

Tendo como início, a expressão "R\\\$[0-9]", precisamos permitir uma quantidade variável de dígitos, mais especificamente na parte "[0-9]". Em ocasiões como essa, nós podemos utilizar os *metacharacters* do tipo quantificadores, que incluem os caracteres ? (ponto de interrogação), + (sinal de mais), * (asterisco) e {} (par de chaves). Como o próprio nome do tipo dá a entender, esses *metacharacters* buscam delimitar a quantidade de vezes que podemos encontrar um certo caractere em nossa sequência. Em outras palavras, esses *metacharacters* definem o número mínimo e máximo de ocorrências possíveis para um caractere específico de nossa expressão.

Primeiro, o *metacharacter* * representa 0 ocorrências como mínimo e infinitas ocorrências como máximo. Com isso, podemos dizer que o *metacharacter* * significa: “tente encontrar esse caractere, o maior número de vezes possíveis, contudo, está tudo bem se não conseguirmos encontrá-lo em algum lugar”. Logo, a expressão "A6*" nos permite encontrar uma letra “A”, quando acompanhada, por exemplo, pelo “número do diabo” (“A666”), ou por qualquer outra quantidade do número 6, como o texto “A6”, ou “A6666666”. Porém, o *metacharacter* * também nos dá a possibilidade de

Figura 10.5: Expressões regulares sobre valores monetários

Fonte: Elaboração própria do autor.

não encontrarmos o número 6. Por isso, a expressão "A6*" também é capaz de encontrar o texto "Ana Luísa", mesmo que ele não possua um número 6.

Segundo, o *metacharacter* + representa 1 ocorrência como mínimo e infinitas ocorrências como máximo. Por consequência, o *metacharacter* + expressa: "tente encontrar esse caractere pelo menos uma vez!". Como exemplo, a expressão "Isa+" é capaz de encontrar os textos "Isadora", "Isaac Newton" e "Isaaaaaa3210". Mas não é capaz de encontrar o texto "Isótopo", pois esse texto não possui pelo menos um "a" logo após os caracteres "Is".

Terceiro, o *metacharacter* ? representa 0 repetições como mínimo e 1 repetição como máximo. Isto é, o *metacharacter* ? busca tornar um caractere completamente opcional. Em outras palavras, ao conectarmos um caractere ou uma sub-expressão ao *metacharacter* ? estamos dizendo algo como: "se esse caractere for encontrado, ótimo! Se não, sem problemas!". Como exemplo, a expressão "dr?a" busca encontrar uma letra "d" imediatamente seguida pelos caracteres "ra". Mas pelo fato de termos incluído o *metacharacter* ? logo à frente da letra "r", tornamos essa letra opcional. Por isso, a expressão "dr?a" é capaz de encontrar textos como "engendrar", "dragão" ou "dramin", assim como os textos "Adaga" e "reciprocidade".

Quarto, o *metacharacter* {} representa a forma geral de um quantificador. Pois ele nos permite especificar exatamente quais as quantidades mínima e máxima que desejamos para um determinado caractere. Basta preencher o par de chaves com essas duas quantidades, separadas por uma vírgula ({min, max}). Por exemplo, a expressão "31[0-9]{4,5}" é capaz de encontrar um código do

IBGE referente a um município do estado de Minas Gerais (os dígitos 31 representam o código do estado de MG). Esses códigos do IBGE possuem uma versão curta, que pode variar de 2 a 4 dígitos, entretanto, suas versões mais comumente utilizadas são as de 6 e de 7 dígitos. Como exemplo, os códigos 310620 e 3106200 se referem ao município de Belo Horizonte. Com isso, ao estabelecermos 4 e 5 dígitos como os limites do intervalo representado pela sub-expressão `[0-9]{4,5}`, somos capazes de detectar códigos como **310620** e **3106200**, e ao mesmo tempo, descartar códigos como 31062, que possui menos de 4 dígitos após os dígitos 31.

Além disso, vale destacar que o objetivo de qualquer *metacharacter* do tipo quantificador, não é o de determinar o número de vezes que um caractere pode aparecer ao **longo do texto**, mas sim, o número de vezes que um caractere pode ocorrer **em sequência**. Por exemplo, a expressão `"(25){2,3}"` busca detectar um número arbitrário de 25's. Assim sendo, essa expressão é capaz de detectar valores como **25**, **252**, e **2525**, da mesma maneira que o texto "Estive na **25** de Março no último dia 25".

Porém, muitas pessoas interpretam que os dois 25's presentes no texto "Estive na 25 de Março no último dia 25" são detectados pela expressão `"(25){2,3}"`, quando na verdade, apenas o primeiro 25 é localizado. Pois o segundo 25 no texto, se encontra a mais de 20 caracteres a frente do primeiro 25. Logo, ao utilizarmos um *metacharacter* do tipo quantificador, estamos geralmente preocupados com a possibilidade de o mesmo caractere aparecer múltiplas vezes em sequência (um atrás do outro).

Voltando à expressão `"R\\$[0-9]"`, com tudo o que descrevi nos parágrafos anteriores, nós podemos adicionar um `+` logo após `[0-9]`. Dessa maneira, estamos desejando encontrar **pelo menos** um número qualquer entre 0 e 9, logo após o símbolo monetário R\$. Com isso, temos a expressão `"R\\$[0-9]+"`, que é capaz de encontrar tanto "R\$3" quanto "R\$3050".

No entanto, ainda temos a possibilidade de encontrarmos um valor monetário que inclui centavos. Ou seja, podemos encontrar um número que seja seguido por uma vírgula e dois outros dígitos que definem os centavos. Por isso, podemos ainda acrescentar a parte `",[0-9][0-9]"` para captar essa possível parte de nosso valor monetário.

```
vec <- c("8730", "R$21", "R$3120,50", "R$43026", "R$45,10")
teste <- str_detect(vec, "R\\$[0-9]+,[0-9][0-9]")
vec[teste]
```

```
## [1] "R$3120,50" "R$45,10"
```

Porém, repare ainda, que ao adicionarmos a seção `",[0-9][0-9]"`, a nossa expressão regular não é mais capaz de detectar valores que não possuem uma parte para os centavos, como R\$21 e R\$43026. É por essa razão, que eu contornei essa seção por parênteses, e adiciono o *metacharacter* `?` logo em seguida. Pois dessa forma, essa seção passa a ser opcional, de forma que a parte para os centavos deixa de ser obrigatória.

```
vec <- c("8730", "R$21", "R$3120,50", "R$43026", "R$45,10")
```



```
teste <- str_detect(vec, "R\\$[0-9]+(,[0-9][0-9])?")
vec[teste]
```

```
## [1] "R$21"      "R$3120,50" "R$43026"    "R$45,10"
```

10.8.7.1 Conclusão e algumas dicas extras

Recaptulando o que vimos até aqui, temos que os números de ocorrências representados por cada *metacharacter* do tipo “quantificador” são:

- `?`: 0 ou 1 ocorrência.
- `+`: 1 ou mais ocorrências.
- `*`: 0 ou mais ocorrências.
- `{min, max}`: entre min e max ocorrências.

Para além do que ainda não foi comentado nessa seção, você pode utilizar novamente o *metacharacter* `{}`, para especificar um número específico de ocorrências que você deseja para um caractere, ou então, definir apenas o número mínimo ou o número máximo de repetições. Com isso, temos que:

- `{n}`: exatamente n ocorrências.
- `{min,}`: pelo menos min ocorrências.
- `{,max}`: até max ocorrências.

10.8.8 Determinando os limites de uma palavra

Como estabelecemos anteriormente, expressões regulares não tem a capacidade de diferenciar palavras, e muito menos, de identificar os seus limites. Por essa razão, para termos garantia de que vamos encontrar uma palavra específica no resultado de uma expressão regular, precisamos estabelecer limites para a pesquisa.

Na seção sobre **Âncoras (*anchors*)**, utilizamos os *metacharacters* do tipo âncora (`^` e `$`) para estipularmos os limites da palavra a ser pesquisada. Porém, esses *metacharacters* **não foram criados para esse objetivo**. Essa afirmação fica clara, ao retornarmos ao exemplo utilizado na seção supracitada.

Naquela ocasião, estávamos tentando encontrar todos os textos contidos no vetor `text`, que possuíssem a palavra “emissão”. Entretanto, ao utilizarmos a expressão `^emissão$`, fomos capazes de encontrar apenas o primeiro elemento de `text`. Sendo que, de acordo com o nosso objetivo, também desejamos localizar o quarto, quinto e sexto elementos de `text`. Pois eles também possuem a palavra “emissão” em alguma instância.

```
text <- c(
  "emissão",
```



```

"A Ford Brasil executou recentemente uma demissão em massa",
"remissão",
"Para mais, a emissão de CO2 cresceu no Brasil",
"emissão de SO2 faz parte do processo",
"A firma foi processada por tal emissão"
)

teste <- str_detect(text, "^emissão$")
text[teste]

## [1] "emissão"

```

Por isso, precisamos de uma nova estratégia para estipularmos esses limites. Lembre-se que uma expressão regular, nada mais é, do que uma descrição concisa de uma sequência específica de caracteres. Logo, precisamos encontrar alguma forma de descrevermos os caracteres que podem representar os limites de uma palavra.

Todavia, para isso, nós precisamos primeiro identificar o que é o limite de uma palavra. Ou redefinindo a questão, o que exatamente separa uma palavra das demais? Com algum tempo de reflexão, você talvez chegue a conclusão de que o que separa uma palavra da outra, são espaços em branco, ou então, símbolos de pontuação, como um ponto final, ou uma vírgula.

Portanto, precisamos incluir em ambos os lados da palavra “emissão” alguma expressão que possa descrever especificamente esses caracteres, como a expressão "`(\\s|[,!.,?])`". Repare que o par de parênteses nessa expressão, busca apenas limitar o alcance do *metacharacter* |, que está separando duas alternativas, ou duas sub-expressões (`\\s` e `[,!.,?]`) que podem descrever os caracteres de nosso interesse. Lembre-se que o termo `\\s` representa o comando `\s`, que é um atalho para uma classe de caracteres que busca localizar qualquer tipo de espaço em branco.

```

teste <- str_detect(text, "(\\s|[,!.,?])emissão(\\s|[,!.,?])")
text[teste]

## [1] "Para mais, a emissão de CO2 cresceu no Brasil"

```

Contudo, perceba acima, que o resultado de nossa pesquisa continua incorreta. Há algum outro detalhe que estamos esquecendo de incluir em nossa expressão. Pois dessa vez, apenas o quarto elemento de `text` foi retornado. Isso ocorre, porque estamos ignorando a possibilidade da palavra de nosso interesse, ser a responsável por iniciar ou terminar uma linha do texto. Logo, precisamos acrescentar os *metacharacters* `^` e `$`, em nossa descrição dos limites de uma palavra. Com isso, temos as expressões `(^|\\s|[,!.,?])` e `($|\\s|[,!.,?])`.

```

teste <- str_detect(text, "(^|\\s|[,!.,?])emissão($|\\s|[,!.,?])")
text[teste]

```

```
## [1] "emissão"
## [2] "Para mais, a emissão de CO2 cresceu no Brasil"
## [3] "emissão de SO2 faz parte do processo"
## [4] "A firma foi processada por tal emissão"
```

Agora sim, fomos capazes de encontrar todos os textos presentes em `text` que possuem a palavra “emissão”.

10.8.8.1 Conclusão e algumas dicas extras

Para pesquisarmos por palavras específicas em uma expressão regular, nós precisamos incluir uma descrição dos caracteres que podem representar os limites físicos de uma palavra. Os limites de uma palavra geralmente assumem no formato de:

- Um espaço em branco (descrito por `[ ]` ou por `\\s`).
- Pontuações (vírgulas, ponto final, etc.; descrito por `[! . , ? ]`).
- Início ou o fim de uma linha (descrito por `^` e `$`).

Vale ainda destacar, o fato de que o R nos oferece um atalho para indicarmos o limite de uma palavra, que se trata do comando `\\b`, ou como deve ser escrito no R, `\\b`. Consequentemente, se você desejasse encontrar, por exemplo, a palavra “camisa”, você poderia utilizar a expressão `"\\bcamisa\\b"`.

10.8.9 Agrupamentos e *backreferencing*

Em vários estilos de expressões regulares, parênteses são capazes de “lembrar” o texto encontrado pela sub-expressão que eles encapsulam (FRIEDL, 2006, p.21). Em expressões regulares, esse mecanismo é comumente denominado de *backreferencing*.

Em resumo, ao contornarmos uma sub-expressão com um par de parênteses, nós estamos formando um “grupo”, e qualquer que seja o pedaço de texto encontrado especificamente por esse grupo, nós somos capazes de reutilizar esse texto dentro da mesma expressão que o localizou, por meio de suas referências numéricas, como `\\1`, `\\2`, `\\3`, e assim por diante. Entenda que essas referências numéricas, nada mais são do que índices de cada par de parênteses, ou de cada grupo presente em sua expressão regular. Logo, o índice `\\1` se refere ao texto localizado pela sub-expressão do primeiro par de parênteses. Já o índice `\\2`, se refere ao texto descrito pela sub-expressão do segundo par de parênteses. E assim segue.

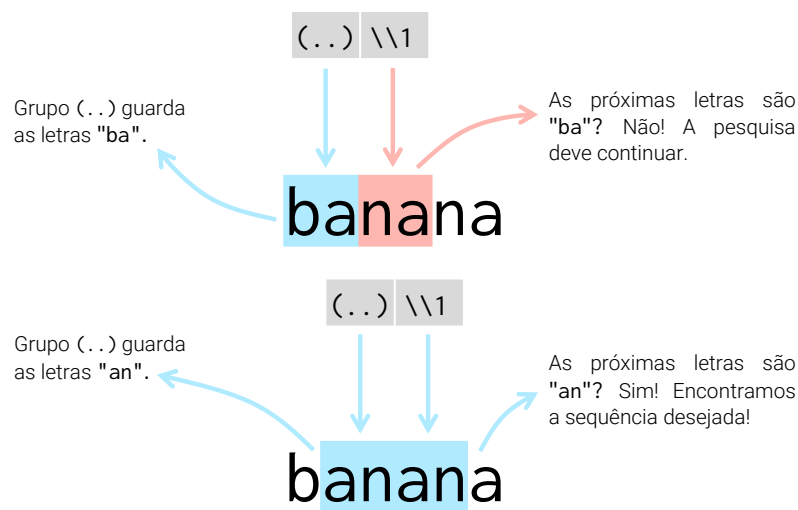
O exemplo clássico desse tipo de operação, está na localização de letras ou palavras repetidas, em uma determinada cadeia de texto. Por exemplo, a expressão abaixo `"(\\. )\\1"`, citada por Wickham e Grolemond (2017, p. 206), busca encontrar dentro do vetor `fruit`, alguma palavra que possua um par de letras repetido em sequência. Por isso, palavras como “banana” e “coconut” são encontradas por essa expressão.

```
teste <- str_detect(fruit, "(.)\\1")
fruit[teste]

## [1] "banana"      "coconut"     "cucumber"    "jujube"      "papaya"
## [6] "salal berry"
```

Portanto, dentro da expressão "(.)\\1", o índice \\1 está fazendo alusão ao par de caracteres encontrados pela sub-expressão "(.)". Entretanto, é importante que você tenha cuidado aqui. Pois o índice \\1 **não corresponde** à expressão regular "(.)". Ou seja, a expressão "(.)\\1" **não é equivalente** à expressão "(.)(.)". Perceba que caso essas expressões fossem iguais, estaríamos simplesmente pesquisando por uma sequência de 4 caracteres quaisquer. Logo, não apenas a correspondência detectada pela expressão seria “**banana**”, mas também, palavras como “**raspberry**” e “**pomegranate**” estariam inclusas no resultado (o que não ocorre acima).

Figura 10.6: Como o mecanismo de pesquisa funciona quando utilizamos *backreferencing*



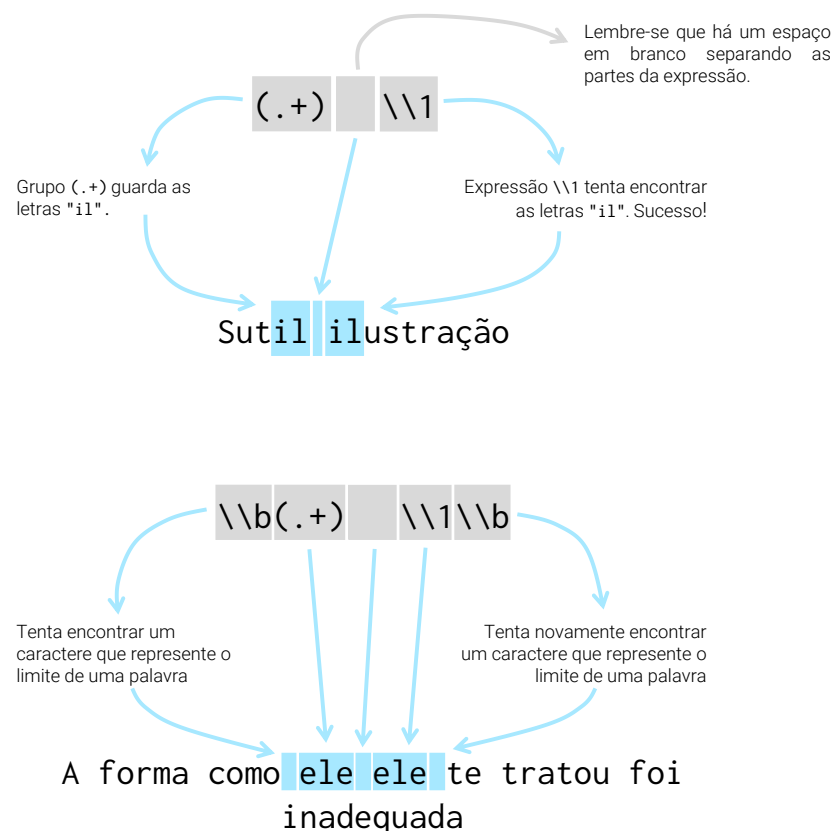
Fonte: Elaboração própria do autor.

Por isso, utilizamos o índice \\1 quando desejamos encontrar o mesmo pedaço de texto, ou a mesma sequência de caracteres encontrada pelo grupo a que se refere. Com isso, *backreferencing* se torna um mecanismo útil quando ainda não conhecemos o texto repetido a ser encontrado, ou quando sabemos que esse texto pode variar violentamente ao longo do texto. Por exemplo, suponha que exista em nosso texto, três casos de palavras repetidas (“que que”, “da da” e “ele ele”). Para encontrar esses casos, você talvez tentaria a expressão "\\bque que\\b|\\bda da\\b|\\bele ele\\b". Porém, seria muito desgastante escrever uma alternativa para cada variação.

Por esse motivo, poderíamos resumir esses casos com o uso de *backreferencing*. Um exemplo de expressão seria "\\b(.+) \\1\\b". Dessa forma, a expressão "(.+) " busca encontrar uma sequência qualquer de caracteres, e o índice \\1 tenta encontrar essa mesma sequência de caracteres logo após um espaço em branco.

Além desses pontos, repare que utilizamos o atalho \\b (que apresentamos ao final da seção anterior) para definirmos os limites de palavras, ao longo de várias dessas expressões. Se você está querendo descobrir palavras repetidas em seus textos, você com certeza deseja definir esses limites de palavras. Pois caso você não o faça, as repetições de uma sequência específica de caracteres, pelas quais você estaria pesquisando, poderiam ocorrer em qualquer lugar e invadir o espaço de outras palavras.

Figura 10.7: A importância de se incluir os limites de palavras em pesquisas que utilizam *backreferencing*



Fonte: Elaboração própria do autor.

Isso significa, que a expressão "(que) \\1" seria capaz de encontrar o texto "A imagem de Naka**que** **que**ima em meu corpo", ou o texto "É claro **que** **que**ro!". Ampliando esse exemplo para uma

expressão mais geral, poderíamos rapidamente realizar que a expressão "(.+) \\1" seria capaz de encontrar textos como “Sutil **ilustração**”, assim como “fez-se **engendrado**”. Dessa forma, o atalho \\b impõe limites a nossa pesquisa, que evitam esse tipo de inconveniência.

10.8.10 Mais sobre padrões

Mesmo estando presente em diversos programas e linguagens, as expressões regulares possuem certa variabilidade, ou apresentam diferentes “gostos” ou “estilos” em cada uma dessas plataformas. Ou seja, as linguagens JavaScript, Perl, Python e R, oferecem um mecanismo próprio de expressões regulares, porém, a forma como esse mecanismo é implementado e quais são as funcionalidades que ele permite, variam em cada linguagem. Por exemplo, em Perl, você normalmente contorna a sua expressão regular, por barras inclinadas a direita, acompanhadas de um “m” na primeira barra (m/expressao/). Já na no R, não se utiliza tal cápsula, e apenas a expressão regular é fornecida.

Por padrão, as funções da família grep() adotam o padrão POSIX 1003.2 de expressões regulares estendidas (*extended regular expressions*), que é equivalente ao estilo oferecido pelo programa egrep. Entretanto, essas funções também permitem o uso de expressões regulares no estilo adotado pela linguagem Perl. Basta configurar o seu argumento perl para TRUE.

Por outro lado, as funções do pacote stringr utilizam as bibliotecas em C do projeto ICU (*International Components for Unicode*). O estilo de expressões regulares oferecido por essa biblioteca é muito inspirado no estilo adotado pelo linguagem Perl e, por essa razão, está um pouco mais próximo do estilo tradicionalmente adotado pela grande maioria das linguagens de programação que oferecem esse recurso. Para mais detalhes sobre essa biblioteca, além de uma lista bem útil de todos os *metacharacters* disponíveis, você pode consultar o [site do projeto](#).

10.9 Substituindo partes de um texto com str_replace()

A função str_replace() e sua variante str_replace_all(), lhe permite aplicar uma expressão regular sobre o seu texto, e substituir a área encontrada (ou áreas encontradas) por um novo valor de seu interesse. Por exemplo, se eu possuo o conjunto de palavras abaixo, e desejo substituir qualquer vogal por um *underline*, eu precisaria do seguinte comando.

```
palavras <- c("arquivo", "estante", "livro", "estiagem",
             "dinheiro", "paz")

palavras <- str_replace(palavras, "[aeiou]", "_")

palavras

## [1] "_rquivo"  "_stante"  "l_vro"    "_stiagem" "d_nheiro" "p_z"
```

Entretanto, perceba acima, que apenas a primeira vogal é alterada. Isso não apenas é um comportamento natural da função str_replace(), mas também é um padrão adotado por muitos dos sistemas

de expressão regular. Como foi colocado por Friedl (2006, p. 148): “any match that begins earlier (leftmost) in the string is always preferred over any plausible match that begins later”. Com isso, o autor quis destacar que o ato de parar a pesquisa na primeira correspondência encontrada, faz parte dos princípios de muitas expressões regulares.

Porém, em muitos momentos, haverá a necessidade de sobrepor esse comportamento, de forma que a sua expressão possa encontrar todas as correspondências presentes em um *string*. Por esse motivo, o pacote *stringr* oferece diversas funções variantes que terminam com o padrão **_all()*. Essas funções buscam justamente solucionar esse problema e, por isso, aplicam a expressão regular sobre todo o texto, com o objetivo de encontrar o maior número possível de correspondências.

Portanto, ao empregarmos a variante *str_replace_all()*, desejamos substituir todas as correspondências encontradas por uma expressão regular em cada *string*, por um novo valor textual. Veja que o exemplo abaixo é praticamente idêntico ao anterior, apenas a função *str_replace()* foi alterada para *str_replace_all()*.

```
palavras <- str_replace_all(palavras, "[aeiou]", "_")
```

```
palavras
```

```
## [1] "_rq__v_" "_st_nt_" "l_vr_" "_st__g_m" "d_nh__r_" "p_z"
```

Como um outro exemplo, poderíamos simular o trabalho executado pela função *str_trim()*, com as funções *str_replace()* e *str_replace_all()*. O comando *str_replace(vec, "^()+",)* estaria procurando por qualquer linha que se inicia por uma quantidade *y* (sendo *y* > 0) de espaços em branco, e substituindo esses espaços por nada (""). Dessa maneira, este comando equivale à *str_trim(vec, side = "left")*. Já o comando *str_replace_all(vec, "^()+|(\)+\$",)*, buscaria qualquer linha que se inicia ou termina por uma quantidade *x* de espaços em branco, e em seguida, substituiria esses espaços por nada. Sendo assim, esse comando equivale à *str_trim(vec, side = "both")*.

```
vec <- c(
  " Russo é a língua oficial da Rússia ",
  " Japão se encontra na Ásia",
  "Português nunca foi tão difícil! ",
  " 224,90 ")
```

```
str_replace(vec, "^( )+", "")
```

```
## [1] "Russo é a língua oficial da Rússia "
## [2] "Japão se encontra na Ásia"
## [3] "Português nunca foi tão difícil! "
## [4] "224,90 "
```

```
str_replace_all(vec, "^ ( )+| ( )+$", "")
```

```
## [1] "Russo é a língua oficial da Rússia"
## [2] "Japão se encontra na Ásia"
## [3] "Português nunca foi tão difícil!"
## [4] "224,90"
```

Para mais, *backreferencing* se torna uma ferramenta extremamente útil em conjunto com `str_replace()`. Por exemplo, suponha que você tenha se esquecido de adicionar o símbolo da moeda brasileira em cada valor numérico. Com a expressão regular "`([0-9]+(,[0-9]+)?)`" podemos encontrar esses valores numéricos. Repare que toda a expressão está contornada por parênteses, logo, todo o número é salvo para o índice `\\1`. Dessa maneira, basta antecedermos esse índice pelo símbolo que desejamos inserir ("`R$\\1`").

```
vec <- c("0 litro de leite custa 3,50", "0 ingresso foi caro. Mais de 500 reais!",
        "230015")
```

```
str_replace(vec, "([0-9]+(,[0-9]+)?)", "R$\\1")
```

```
## [1] "0 litro de leite custa R$3,50"
## [2] "0 ingresso foi caro. Mais de R$500 reais!"
## [3] "R$230015"
```

10.10 Dividindo *strings* com `str_split()`

Você também pode utilizar uma expressão regular para detectar “pontos de quebra” em uma cadeia de texto e, em seguida, quebrar essa cadeia nesses pontos determinados. Repare no exemplo abaixo, que a função `str_split()` nos retorna como resultado, uma lista de vetores, onde cada elemento dessa lista, contém os “pedaços” de cada elemento do vetor original (`vec`). Logo, se você está aplicando `str_split()` sobre um vetor com 34 elementos, você terá uma lista com 34 elementos em seu produto final.

```
vec <- c(
  "1 : 2 : 3 : 4 : 5 : 6 : 7",
  "Faria, Pedro Duarte : 1290321_1",
  "Objeto não localizado : 10_0x341167",
  "A732 : B3 : 24 : C1 : 90 : 89 : QUA : ABD : AQZ29 : C11 : 01ER"
)
```

```
str_split(vec, " : ")
```

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

```
##
## [[2]]
## [1] "Faria, Pedro Duarte" "1290321_1"
##
## [[3]]
## [1] "Objeto não localizado" "10_0x341167"
##
## [[4]]
## [1] "A732" "B3" "24" "C1" "90" "89" "QUA" "ABD"
## [9] "AQZ29" "C11" "01ER"
```

Contudo, a depender do que você planeja fazer em seguida, pode ser difícil trabalhar com uma lista. Por isso, a função `str_split()` nos oferece o argumento `simplify`, no qual podemos requisitar a função que simplifique o resultado para uma matriz.

```
str_split(vec, " : ", simplify = TRUE)
```

```
##      [,1]      [,2]      [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] "1"      "2"      "3"  "4"  "5"  "6"  "7"  ""
## [2,] "Faria, Pedro Duarte" "1290321_1" "" "" "" "" ""
## [3,] "Objeto não localizado" "10_0x341167" "" "" "" "" ""
## [4,] "A732"      "B3"      "24" "C1" "90" "89" "QUA" "ABD"
##      [,9]  [,10] [,11]
## [1,] ""    ""    ""
## [2,] ""    ""    ""
## [3,] ""    ""    ""
## [4,] "AQZ29" "C11" "01ER"
```

10.11 Extraindo apenas a correspondência de sua expressão regular com `str_extract()`

Assim como substituir suas correspondências por novos valores, você também tem a capacidade de extrair essas correspondências isoladamente, por meio da função `str_extract()`. Essa funcionalidade se torna extremamente importante quando não apenas a estrutura de cada elemento de seu vetor difere, mas também, quando a posição de seu alvo ao longo da cadeia de texto varia. Essas características tornam impossível a extração de nosso alvo com a função `str_sub()` (que apresentamos anteriormente), que se baseia diretamente na posição dos caracteres ao longo do texto.

Por isso, a melhor alternativa para superarmos esse empecilho, é empregar uma expressão regular que possa detectar os nossos alvos e, com isso, extraí-los por meio da função `str_extract()`. Como exemplo, podemos extrair todos os anos presentes em cada elemento do vetor `per`, através do seguinte comando:


```
per <- c("Janeiro_2020", "Visitei Pará de Minas em Fevereiro de 2019",
        "2020 foi um ano terrível", "O Brasil era a 11ª economia do mundo em 2005")
```

```
str_extract(per, "\\d{4}")
```

```
## [1] "2020" "2019" "2020" "2005"
```

Ou melhor, podemos colocar o texto original e a parte extraída em uma tabela:

```
tibble(
  text = per,
  ano = str_extract(per, "\\d{4}")
)
```

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

```
##   text                                ano
##   <chr>                             <chr>
## 1 Janeiro_2020                      2020
## 2 Visitei Pará de Minas em Fevereiro de 2019  2019
## 3 2020 foi um ano terrível            2020
## 4 O Brasil era a 11ª economia do mundo em 2005 2005
```

Assim como `str_replace()`, `str_extract()` é capaz de extrair apenas a primeira correspondência encontrada por sua expressão regular. Por esse motivo, você irá precisar de sua variante, `str_extract_all()`, em todas as ocasiões em que você tiver mais de um alvo a ser extraído em cada texto. Por exemplo, podemos extrair o valor de cada medida presente em medidas, por meio da expressão `"([0-9]+)([.][0-9]+)?"`.

```
### Largura X Altura X Profundidade (Peso, Classe)
```

```
medidas <- c(
  "8.15 m X 2.23 m X 4.5 m (240 Kg, B)",
  "1.14 m X 3.1 m X 0.9 m (15 Kg, A)",
  "4.98 m X 9.2 m X 5.25 m (120 Kg, A)",
  "3.14 m X 3.89 m X 3.41 m (86 Kg, C)"
)
```

```
tab <- str_extract_all(
  medidas,
  "([0-9]+)([.][0-9]+)?",
  simplify = TRUE
)
```

```
colnames(tab) <- c(
  "Largura", "Altura", "Profundidade", "Peso"
)
```

tab

##		Largura	Altura	Profundidade	Peso
##	[1,]	"8.15"	"2.23"	"4.5"	"240"
##	[2,]	"1.14"	"3.1"	"0.9"	"15"
##	[3,]	"4.98"	"9.2"	"5.25"	"120"
##	[4,]	"3.14"	"3.89"	"3.41"	"86"

Exercícios

Exercício 1

Em cada item dessa questão, você deve criar uma expressão regular que represente a sequência de caracteres descrita no enunciado. Em seguida, você deve aplicar essa expressão regular sobre o vetor `words`, com o objetivo de extrair todas as palavras desse vetor que se encaixam nessa determinada expressão. O vetor `words` advém do pacote `stringr`, logo, se você conseguiu chamar por esse pacote em sua sessão através do comando `library()`, você já possui acesso a esse vetor.

- A) Um “b” ou “c” seguidos de um “a” e um “l”.
- B) Um caractere qualquer (exceto a letra “a”) imediatamente seguido por um “c”, que por sua vez, é seguido pelo final do *string*.
- C) A sequência “s-p-a-c-e” de caracteres, ou, um “e” imediatamente seguido por duas letras “s”, que por sua vez são seguidos imediatamente pelo final da linha.
- D) Crie uma expressão regular que possa encontrar todas as palavras presentes em `words` que contém exatos 3 caracteres. Você pode solucionar essa questão com a função `str_length()`. **Porém, você deve utilizar uma expressão regular para encontrar essas palavras de 3 caracteres**, portanto, esqueça momentaneamente que a função `str_length()` existe¹.

Exercício 2

Os itens desta questão vão trabalhar com o vetor `compras`. Para importar esse vetor para a sua sessão do R, copie e cole os comandos abaixo em seu console. Como você pode ver abaixo, cada elemento do vetor `compras` contém uma *string* bastante longa, tão longa que fui obrigado a utilizar a função `str_trunc()` para cortar parte do texto e apresentar abaixo apenas os 50 primeiros caracteres de cada *string*. Dentro de cada *string*, você possui um conjunto de dados referentes a uma compra realizada em uma loja durante o ano de 2020.

```
library(tidyverse)

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

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

str_trunc(head(compras), width = 50, ellipsis = "~")

## [1] "Márcio390.287.917-210akqzS2tk$URMcL0k5Q2356772.25~"
## [2] "Igor944.236.416-254tLo8&S9WtXg05fsdU2188525.212/0~"
## [3] "Márcio395.304.955-57pfwji9Z4Q6dZxSWZV7#7Z$J218160~"
```

¹Este exercício foi diretamente extraído da obra de Wickham e Grolemund (2017, p. 203)

```
## [4] "Isabela322.900.842-74K5D6b$xAnY&QJ1$XQzE2f1554399~"
## [5] "Álvaro475.767.740-583WWonElfbisKD1GiIVS225066.161~"
## [6] "Rafael031.357.966-89b0zZ7#2JBcsd!sWzaeNY1866117.7~"
```

- A) Como você pôde ver acima, os dados estão misturados em cada *string*. Em outras palavras, a loja que coletou esses dados não se preocupou em utilizar um separador especial para separar as variáveis em diferentes colunas. Agora, eles estão todos juntos, um do lado do outro, em uma única coluna.

Em resumo, cada *string* guarda as informações de 7 variáveis diferentes: nome do consumidor; CPF do consumidor; código de identificação da venda; código de identificação do produto comprado; valor pago por unidade; quantidade adquirida; horário da compra. Precisamente nessa ordem. Como um guia, temos as figuras 10.8 e 10.9 abaixo. Cada figura apresenta uma “metade” específica do primeiro *string* presente no vetor *compras* (o *string* é muito grande, por isso, optou-se por dividi-lo em duas figuras). Cada figura, busca descrever a estrutura seguida por cada *string* do vetor *compras*.

Figura 10.8: Descrição dos 39 primeiros caracteres de cada *string* presente em *compras*

1	2	3
Márcio	390.287.917-21	0akqzS2tk\$URMcL0k5Q
1	2	3
Variável: Nome do consumidor. Comprimento: Varia. Contém: letras maiúsculas e minúsculas.	Variável: CPF do consumidor. Comprimento: 15 caracteres. Contém: letras maiúsculas e minúsculas.	Variável: Código de identificação da venda. Comprimento: Entre 16 e 24 caracteres. Contém: pode conter letras maiúsculas e minúsculas, números de 0 a 9, além dos seguintes símbolos: !#\$*&.

Fonte: Elaboração própria do autor.

O seu trabalho é utilizar as ferramentas que você viu nesse capítulo, para extrair essas 7 variáveis e aloca-las em colunas separadas de um *data.frame*. Esse não é um exercício muito simples, mas ele também transmite certa realidade. Há diversas bases de dados e análises reais na indústria, que exigem um uso intensivo de ferramentas de extração e localização de texto, como é o caso desse exercício sobre o vetor *compras*.

Para realizar essa atividade, você não precisa necessariamente utilizar apenas expressões re-

Figura 10.9: Descrição dos 43 últimos caracteres de cada *string* presente em compras

1	2	3	4
2356	772.25	12/04/2020 06:17:16 -033	unidades

1

Variável: Código do produto.
Comprimento: 4 caracteres.
Contém: apenas números.

2

Variável: Valor por unidade do produto.
Comprimento: Varia.
Contém: números e um ponto que marca a casa decimal.

3

Variável: Horário da compra
Comprimento: 23 caracteres.
Contém: números, além da barra /, dois pontos :, e um sinal de menos -.

4

Variável: Horário da compra
Comprimento: 23 caracteres.
Contém: números, além da barra /, dois pontos :, e um sinal de menos -.

Fonte: Elaboração própria do autor.

gulares por todo o caminho. Dado a complexidade desses *strings*, é interessante e, até mais simples, que você misture um pouco suas técnicas, ao trabalhar com partes (ou *subsets*) específicos dos *strings* com `str_length()` e `str_sub()` e, em seguida, aplicar expressões regulares sobre as partes restantes dos *strings*.

Caso você opte por utilizar uma única expressão regular para resolver esse item, é fundamental que você compreenda bem **como** os valores de cada variável podem variar em cada *string*. Em outras palavras, para que você seja capaz de descrever, com precisão, cada parte da sequência de caracteres que compõe esses *strings*, você precisa saber, por exemplo: quais caracteres podem aparecer, na parte que apresenta o código de identificação da venda; ou ainda, quantos dígitos são permitidos no campo do valor unitário do produto? Para ter essa compreensão, leia atentamente às figuras 10.8 e 10.9.

- B) Volte ao vetor *compras* e extraia de cada *string*, apenas a parte correspondente à data e horário da compra. Com esses valores em mãos, tente capturar o dia de cada data, por último, realize uma contagem sobre esses dias, e descubra o dia do mês em que essa loja possui o maior número de vendas.
- C) Selecione os 3 primeiros dígitos do CPF do consumidor de cada *string*.

Capítulo 11

Introduzindo fatores (factor's) com forcats

11.1 Introdução e pré-requisitos

No capítulo de [Fundamentos da Linguagem R](#), introduzimos os 4 tipos básicos de dados disponíveis no R, sendo eles: integer; double; character; e logical. Entretanto, também destacamos que outros tipos de dados “mais complexos” estão presentes na linguagem R, e, que eles serão tão importantes quanto os tipos básicos em suas análises.

Os exemplos mais importantes desses tipos são os fatores (factor) e as variáveis de tempo, isto é, datas e horários (Date ou POSIXct). Neste capítulo, vamos focar a discussão no tipo factor, e, no próximo capítulo, discutiremos os tipos referentes às variáveis de tempo.

Parte dos exemplos deste capítulo, envolvem o uso de funções do pacote forcats, portanto, não se esqueça de instalar esse pacote (com o comando `install.packages()`), e, logo depois, chamar pelo pacote para a sua sessão (com o comando `library()`). O pacote forcats está incluso no pacote tidyverse, e, por isso, o tidyverse representa um caminho alternativo para você acessar as funções deste pacote.

```
library(forcats)
## Ou
library(tidyverse)
```

11.2 O que são fatores ?

Um fator (ou factor) é um tipo de dado do R desenvolvido para o trabalho com **variáveis categóricas**, ou **variáveis qualitativas**. Ou seja, o tipo de dado factor lhe permite armazenar características e qualidades que um indivíduo carrega, ou de outra forma, qual a “categoria” ou grupo em que cada indivíduo de sua tabela se encaixa.

O sexo e a cor de pele são dois exemplos clássicos de variáveis qualitativas, pois elas identificam uma característica física do indivíduo. Características essas que determinam se o indivíduo pertence

ou não a um grupo específico de pessoas (mulheres pardas, homens brancos, etc.). A faixa etária é um outro exemplo muito comum, sendo uma variável que busca separar indivíduos em vários grupos de acordo com as suas idades.

Entretanto, para além de características e categorias, também podemos identificar uma variável categórica, ao percebermos se essa variável pode (ou deve) assumir um conjunto muito específico e muito bem definido de valores (R CORE TEAM, 2020b, p. 8). Por exemplo, uma variável que apresente o sexo de uma pessoa pode assumir apenas dois valores diferentes (Homem ou Mulher; Masculino ou Feminino; H ou M; ou alguma outra variação desses valores). Pode haver ainda, a necessidade de incluir um terceiro valor para casos especiais, como “Indefinido”, mas em geral, o sexo assume apenas os dois valores supracitados¹. Como um outro exemplo, uma variável que guarda o mês do ano ao qual os dados de sua tabela se referem pode assumir apenas doze valores diferentes (Janeiro, Fevereiro, Março, ..., Novembro, Dezembro), logo, essa também é uma variável categórica sob essa perspectiva.

11.3 Como construir um fator

Suponha que você tenha questionado o sexo de várias pessoas, e anotado as suas respostas no vetor abaixo (entrevista):

```
entrevista <- c("Mulher", "Homem", "Homem", "Mulher", "Mum")
```

Se você deseja transformar esse vetor acima (que no momento é um vetor do tipo character) em um vetor do tipo factor, você deve primeiro pensar sobre o atributo levels que será utilizado neste vetor. Ou seja, **todo objeto do tipo factor no R possui um atributo chamado levels**, que representa o conjunto de valores que a variável em questão pode assumir. Como estamos anotando o sexo de algumas pessoas entrevistadas, sabemos que essa variável pode assumir apenas dois valores diferentes. Eu crio o vetor abaixo (niveis_sexo) com o objetivo de guardar essas informações.

```
niveis_sexo <- c("Homem", "Mulher")
```

Agora que temos o vetor com a informação original (entrevista) e um vetor com os níveis, ou, os valores permitidos para essa variável (niveis_sexo), podemos criar o nosso fator através da função factor().

```
vec_fator <- factor(entrevista, levels = niveis_sexo)
vec_fator
```

¹Pode haver certa confusão entre sexo e gênero aqui. O sexo se refere às características físicas e biológicas do corpo, e essas características podem identificar uma pessoa como Homem ou Mulher. Já o gênero, está muito mais relacionado à cultura e a forma como um indivíduo se identifica como ser. Logo, se nossa variável identificasse o gênero de uma pessoa, haveria muito mais possibilidades do que a simples divisão entre Homem e Mulher.


```
## [1] Mulher Homem  Homem  Mulher <NA>
## Levels: Homem Mulher
```

Perceba acima, que ao chamarmos pelo novo fator criado, os níveis da variável (atributo `levels`) são mostrados logo abaixo dos valores armazenados. Repare também, que todos os valores presentes no vetor original (entrevista) e que estejam fora dos níveis da variável (`niveis_sexo`) são silenciosamente convertidos para valores NA. Isto é, qualquer valor que esteja minimamente divergente dos valores presentes em `levels`, ou que contenha algum erro ortográfico, será convertido para um valor NA.

Você sempre pode acessar os níveis (isto é, o atributo `levels`) de um fator por meio da função `levels()`. Basta aplicá-la diretamente sobre o fator, que um vetor contendo esses níveis será retornado para você.

```
levels(vec_fator)
```

```
## [1] "Homem"  "Mulher"
```

Vale destacar, que para o R, um vetor do tipo `factor`, é na verdade, um vetor do tipo `integer` que carrega uma classe `factor`, e que possui um atributo chamado `levels`. Esse é um dos principais motivos pelos quais os tipos `factor`, `Date` e `POSIXct` são caracterizados como tipos “mais complexos” da linguagem R. Pois esses tipos são construídos a partir dos quatro tipos básicos, mas eles também acumulam novas características ou propriedades que não estão presentes nesses tipos básicos.

No caso do tipo `factor`, ele é construído a partir do tipo `integer` devido a forma como o R guarda os valores presentes em um vetor do tipo `factor` (R CORE TEAM, 2020b, p. 8). Por exemplo, os valores “Homem” e “Mulher” do vetor `vec_fator` acima, são guardados pelo R como valores 1 e 2, e são posteriormente traduzidos como valores “Homem” e “Mulher” quando chamamos pelo vetor `vec_fator`. Tudo isso ocorre, devido às propriedades e atributos que um vetor do tipo `factor` carrega, e que o diferenciam de um vetor do tipo `integer`.

```
typeof(vec_fator)
```

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

```
class(vec_fator)
```

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

```
attributes(vec_fator)
```

```
## $levels
## [1] "Homem"  "Mulher"
##
## $class
## [1] "factor"
```

11.4 Porque utilizar fatores se eu posso armazenar como texto ?

Você provavelmente está se perguntando qual a necessidade verdadeira dos fatores, levando em conta que você pode utilizar o tipo `character` para armazenar os dados de um variável qualitativa. Wickham e Grolemund (2017, p. 224) nos concede um ótimo exemplo de como um fator pode fazer toda a diferença.

Por exemplo, suponha que você possua o vetor abaixo contendo alguns meses do ano. Em geral, há dois problemas no uso de um vetor do tipo `character` para guardar essas informações.

```
vec <- c("Mar", "Fev", "Jan", "Set", "Out", "Abr")
```

Primeiro, você não está prevenido contra possíveis erros ortográficos. Isso pode ser um problema de pouca importância caso esses dados estejam sendo gerados por uma máquina ou programa, mas ele se torna um problema sério caso você esteja anotando esses valores na mão, ou esteja constantemente corrigindo-os de alguma maneira que seja suscetível ao erro. Logo, se algum mês for incorretamente gravado, nenhum erro ou medida cautelar será acionada pelo R para corrigir esse problema.

Segundo, quando essas informações estão sendo guardadas pelo tipo `character`, o sistema de ordenação utilizado pelo R (ordenação alfabética) é de pouca utilidade. Como você pode ver abaixo, o R acabou colocando o mês de Abril antes dos meses de Fevereiro e Janeiro.

```
sort(vec)
```

```
## [1] "Abr" "Fev" "Jan" "Mar" "Out" "Set"
```

O uso do tipo `factor` consegue resolver ambos desses problemas. Pois você já sabe que qualquer valor disposto em `vec`, que possua algum erro ortográfico em comparação com os meses dispostos no atributo `levels` do fator será automaticamente convertido para um valor `NA`. Além disso, **ao ordenar um objeto do tipo `factor`, o R sempre vai utilizar como referência, a ordem na qual os valores estão apresentados no atributo `levels`.**

Como o vetor `vec` guarda alguns meses do ano, o vetor `meses` abaixo, representa o atributo `levels` do fator a ser criado a partir de `vec`. Lembre-se que, a ordem na qual os meses estão dispostos no atributo `levels`, afeta diretamente a maneira como o R ordena o fator. Logo, a ordem em que você fornece os valores em meses, será a ordem utilizada pelo R ao ordenar os valores de `vec_fator`.

```
meses <- c("Jan", "Fev", "Mar", "Abr", "Mai", "Jun",  
          "Jul", "Ago", "Set", "Out", "Nov", "Dez")
```

```
vec_fator <- factor(vec, levels = meses)  
vec_fator
```

```
## [1] Mar Fev Jan Set Out Abr
## Levels: Jan Fev Mar Abr Mai Jun Jul Ago Set Out Nov Dez
```

```
sort(vec_fator)
```

```
## [1] Jan Fev Mar Abr Set Out
## Levels: Jan Fev Mar Abr Mai Jun Jul Ago Set Out Nov Dez
```

11.5 Não construir o atributo levels é contraintuitivo

Apesar de ser o ideal, você não precisa obrigatoriamente construir o atributo levels ao formar um fator. Pois você tem a opção de delegar esse trabalho para a própria função `factor()`, ao aplicá-la diretamente sobre o seu vetor de interesse.

Porém, ao escolher esse caminho, `factor()` vai extrair todos os valores únicos de seu vetor, e posicioná-los em ordem alfabética no atributo levels. Ou seja, supondo que o seu vetor de interesse se chame `x`, é como se o atributo levels de seu fator, equivalesse ao resultado dos comandos: `unique(x) %>% sort()`; ou de outra forma: `sort(unique(x))`. Veja o exemplo abaixo:

```
v_letras <- c("e", "a", "b", "c", "a", "b", "d")
f <- factor(v_letras)
f
```

```
## [1] e a b c a b d
## Levels: a b c d e
```

```
sort(f)
```

```
## [1] a a b b c d e
## Levels: a b c d e
```

Perceba acima, que tal comportamento de `factor()` torna o uso de fatores, algo inútil ou desnecessário. Pois a ordenação de seu fator será idêntica à ordenação alfabética utilizada sobre um vetor do tipo `character`. Lembre-se que para a ordenação de um fator, é utilizada a ordem na qual os valores são apresentados em `levels()`. Tal ponto pode ser inferido pelo exemplo abaixo, em que a ordenação produzida sobre os valores de `v_letras` é a mesma (em comparação com o resultado acima) quando ela se encontra no tipo `character`.

```
v_letras <- c("e", "a", "b", "c", "a", "b", "d")
typeof(v_letras)
```

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

```
sort(v_letras)
```

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

11.6 Alterando a ordem dos níveis de um fator

Portanto, o sistema de ordenação é um dos principais recursos do tipo `factor` no R, e tal sistema está diretamente conectado com o seu atributo `levels`. Por isso, uma das principais atividades com fatores está na reordenação e do atributo `levels`, ou em sua reatribuição.

11.6.1 A maneira mais simples e direta

A forma mais “simples” de alterarmos esse atributo é redefinindo-o por completo através da função `levels()`. Repare no exemplo abaixo, que apenas a letra “a” foi reposicionada no atributo.

```
levels(f) <- c("b", "c", "d", "e", "a")
sort(f)
```

```
## [1] b b c c d e a
## Levels: b c d e a
```

Tal operação poderia ser realizada de diversas formas. Por exemplo, caso o seu fator possua um número muito grande de níveis, ao invés de reescrevê-los na mão, talvez seja mais rápido utilizar técnicas de *subsetting* para reordenar os níveis da maneira desejada.

```
## Criando um fator com muitos níveis
f <- c("a", "b", "c", "d", "e")
levels(f) <- c(
  "a", "b", "c", "d", "e", "f", "g",
  "h", "i", "j", "k", "l", "m", "n",
  "o", "p", "q", "r", "s", "t", "u",
  "v", "w", "x", "y", "z"
)

## Selecionando os níveis atuais
## e reordenando-os com subsetting
niveis_atuais <- levels(f)
n_niveis <- length(niveis_atuais)
novos_niveis <- niveis_atuais[c(4:2, 5:n_niveis, 1)]

## Redefinindo os níveis do fator
levels(f) <- novos_niveis
f

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

11.6.2 Maneiras alternativas que podem fazer a diferença

O pacote `forcats` oferece várias funções voltadas especificamente para o trabalho com fatores no R. Dentre essas funções, temos a `fct_infreq()`, que lhe permite reordenar o atributo `levels` de acordo com a frequência em que cada nível aparece no vetor (do nível mais frequente para o menos frequente).

```
v_letras <- c("e", "d", "d", "c", "a", "a", "a", "c",
              "b", "d", "d", "e", "d", "a", "d", "c")
f <- factor(v_letras)
fct_infreq(f)
```

```
## [1] e d d c a a a c b d d e d a d c
## Levels: d a c e b
```

Além disso, você também pode estar interessado em ordenar os níveis de um fator, de acordo com a ordem da primeira aparição de cada nível. Para isso, nós podemos utilizar a função `fct_inorder()`. Perceba pelo resultado do exemplo abaixo, que as letras “e”, “d” e “c” antecedem as letras “a” e “b” no atributo `levels` do fator gerado, pois essas letras aparecem primeiro no vetor original.

```
fct_inorder(f)
```

```
## [1] e d d c a a a c b d d e d a d c
## Levels: e d c a b
```

Para mais, haverá momentos em que você deseja ordenar os níveis de seu fator, de acordo com uma segunda variável. Essa situação ocorre principalmente quando o seu fator está incluso em um `data.frame`, junto de várias outras variáveis de seu interesse. Para tal ação, temos a função `fct_reorder()`, que lhe permite fornecer uma segunda variável na qual a ordenação do atributo `levels` será baseada.

Como exemplo, suponha que você possua a seguinte tabela contendo receitas mensais de alguns lojas:

```
unidades <- c("Savassi", "Centro", "Gameleira", "Pampulha")
set.seed(3)
tab <- tibble(
  ano = 2021,
  mes = rep(1:12, each = 4),
  unidade = rep(unidades, times = 12),
  receita = rnorm(48, 17000, 4800)
)
tab <- arrange(tab, mes, unidade)
tab
```

```
## # A tibble: 48 x 4
##   ano   mes unidade  receita
##   <dbl> <int> <chr>    <dbl>
## 1  2021     1 Centro    15596.
## 2  2021     1 Gameleira 18242.
## 3  2021     1 Pampulha 11470.
## 4  2021     1 Savassi 12383.
## 5  2021     2 Centro    17145.
## 6  2021     2 Gameleira 17410.
## 7  2021     2 Pampulha 22360.
## 8  2021     2 Savassi 17940.
## 9  2021     3 Centro    23083.
## 10 2021     3 Gameleira 13425.
## # ... with 38 more rows
```

No exemplo abaixo, ao transformarmos a variável unidade em um fator, os níveis da variável são organizados em ordem alfabética, como era esperado.

```
tab$unidade <- factor(tab$unidade)
tab$unidade
```

```
## [1] Centro   Gameleira Pampulha Savassi  Centro   Gameleira Pampulha
## [8] Savassi  Centro   Gameleira Pampulha Savassi  Centro   Gameleira
## [15] Pampulha Savassi  Centro   Gameleira Pampulha Savassi  Centro
## [22] Gameleira Pampulha Savassi  Centro   Gameleira Pampulha Savassi
## [29] Centro   Gameleira Pampulha Savassi  Centro   Gameleira Pampulha
## [36] Savassi  Centro   Gameleira Pampulha Savassi  Centro   Gameleira
## [43] Pampulha Savassi  Centro   Gameleira Pampulha Savassi
## Levels: Centro Gameleira Pampulha Savassi
```

A função `fct_reorder()` vai sempre ordenar o seu fator de acordo com um sumário, ou alguma estatística descritiva da segunda variável. Por isso, você deve se perguntar qual estatística descritiva você deseja utilizar sobre a segunda variável em questão. Como exemplo, você talvez queira ordenar os níveis de unidade, de acordo com a receita média mensal de cada loja.

Logo, desejamos aplicar uma função de média sobre a variável receita ao longo de cada nível do fator unidade. Por isso, eu forneço a função `mean()` ao argumento `.fun` de `fct_reorder()`. Como podemos ver abaixo, as unidades do Centro e da Savassi possuem receitas médias menores do que unidades da Pampulha e da Gameleira, pois essas unidades se encontram nas primeiras posições do atributo `levels` do fator resultante de `fct_reorder()`. Ou seja, a função `fct_reorder()` utiliza, por padrão, uma ordem crescente no atributo `levels`. Caso você deseje inverter esse comportamento, basta configurar o argumento `.desc` da função para `TRUE`.

```
## Utilize: fct_reorder(unidade, receita, .fun = mean, .desc = TRUE)
## para utilizar uma ordenação crescente no atributo levels
```

```

tab <- tab %>%
  mutate(
    unidade = fct_reorder(unidade, receita, .fun = mean)
  )

tab$unidade

## [1] Centro    Gameleira Pampulha Savassi  Centro    Gameleira Pampulha
## [8] Savassi   Centro    Gameleira Pampulha Savassi   Centro    Gameleira
## [15] Pampulha Savassi   Centro    Gameleira Pampulha Savassi   Centro
## [22] Gameleira Pampulha Savassi   Centro    Gameleira Pampulha Savassi
## [29] Centro    Gameleira Pampulha Savassi   Centro    Gameleira Pampulha
## [36] Savassi   Centro    Gameleira Pampulha Savassi   Centro    Gameleira
## [43] Pampulha Savassi   Centro    Gameleira Pampulha Savassi
## Levels: Centro Savassi Pampulha Gameleira

```

11.7 Reordenando fatores em gráficos

A ordem na qual apresentamos certas informações pode mudar drasticamente não apenas as características físicas e visuais de seu gráfico, mas também, pode afetar e muito a clareza ou a ênfase em certas informações que são cruciais em nosso gráfico. Por essa razão, reordenar variáveis categóricas em seu gráfico pode ser fundamental. Veja o primeiro exemplo abaixo, dado por [Wickham e Grolemund \(2017, p. 228\)](#).

Dentre as funções que mostramos na seção passada, a função `fct_reorder()` é talvez a mais útil delas em gráficos. Por exemplo, no gráfico abaixo, temos certa dificuldade em comparar e, principalmente, classificar os vários tempos médios gastos dentro de cada grupo religioso.

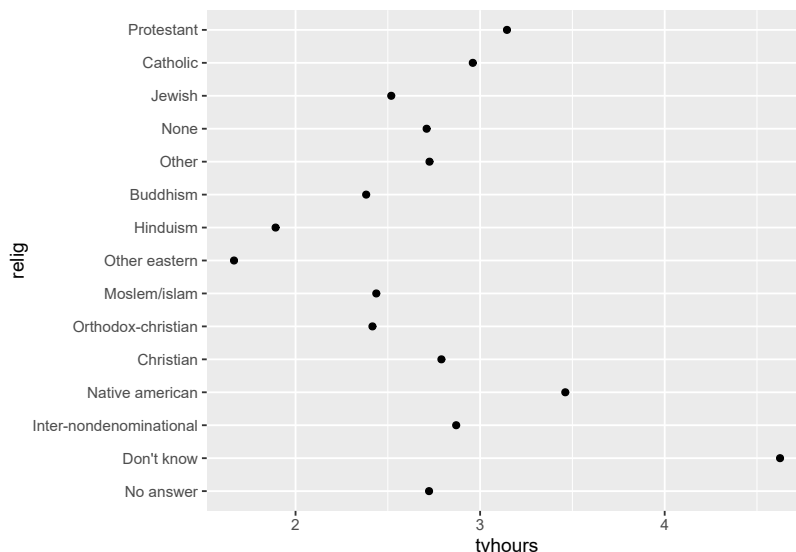
```

relig <- gss_cat %>%
  group_by(relig) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

relig %>%
  ggplot() +
  geom_point(aes(tvhours, relig))

```

Tal problema, pode ser rapidamente resolvido ao aplicarmos a função `fct_reorder()` sobre a variável no eixo y, para que ela seja reordenada de acordo com os valores da variável do eixo x do gráfico. Perceba abaixo, que agora temos uma facilidade muito maior em comparar e classificar os



Fonte: Elaboração própria do autor.

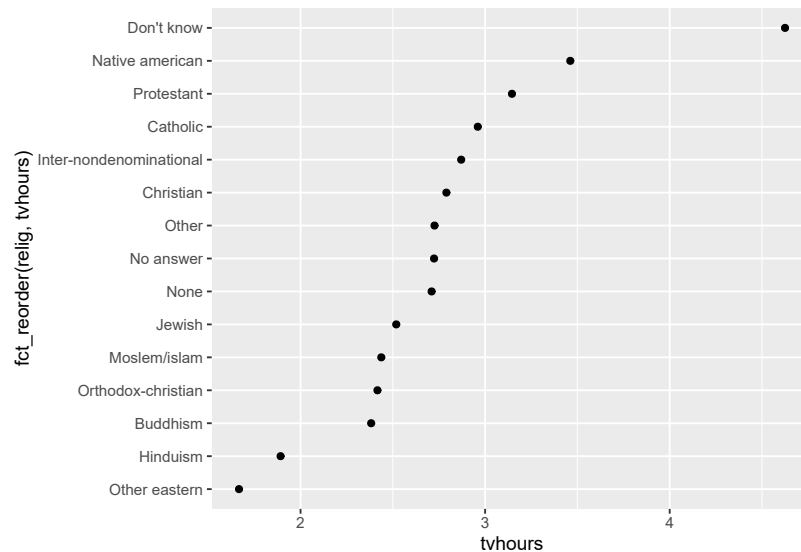
vários tempos médios gastos em cada grupo religioso. Com essa nova ordenação, podemos rapidamente identificar que as pessoas que não sabem (“*Don’t know*”) a sua religião (ou que são ateus), são aquelas que mais gastam seu tempo em frente a uma televisão.

```
relig %>%
  ggplot() +
  geom_point(
    aes(tvhours, fct_reorder(relig, tvhours))
  )
```

Como um outro exemplo, pode haver certas variáveis que não necessitam de uma reordenação acentuada. Além disso, tais variáveis podem possuir uma ordem própria, que não depende de uma segunda variável. Ou seja, essas variáveis podem possuir uma “ordem natural”. Essa característica, torna o uso de `fct_reorder()` inadequado (lembre-se que `fct_reorder()` busca reordenar um fator de acordo com os valores de uma segunda variável).

Por exemplo, se você olhar para o gráfico abaixo, você poderá perceber que temos uma variável de faixa etária no eixo y, e que apenas a faixa de “Menos de 10” está incorretamente posicionada no eixo. Pelo fato das faixas etárias possuírem uma “ordem natural”, isto é, as faixas “mais altas” são aquelas referentes às idades mais elevadas, enquanto as faixas “mais baixas” são aquelas referentes às idades “mais baixas”, não faz sentido reordenarmos essa variável de acordo com os valores de uma segunda variável.

```
github <- "https://raw.githubusercontent.com/pedropark99/"
arquivo <- "Curso-R/master/Dados/datasus.csv"
```

Fonte: Elaboração própria do autor.

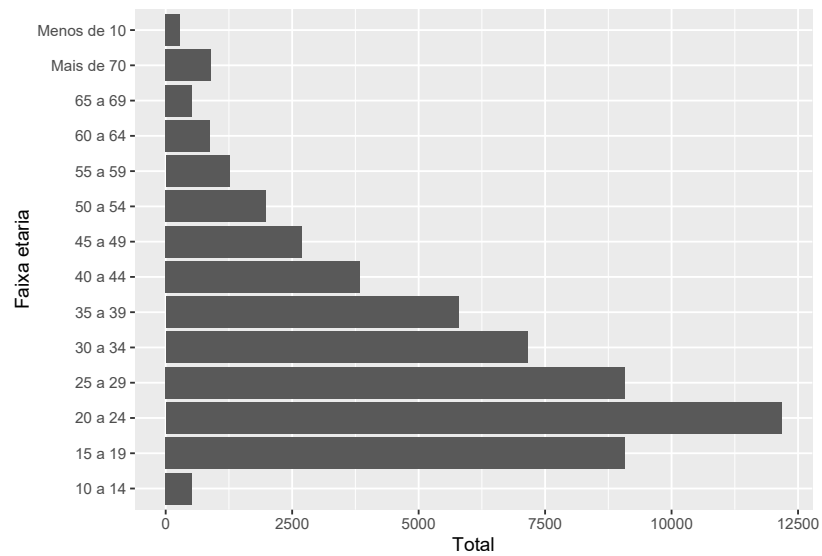
```
datasus <- read_csv2(paste0(github, arquivo))
```

```
totais <- datasus %>%
  group_by(`Faixa etaria`) %>%
  summarise(
    Total = sum(Contagem)
  )
```

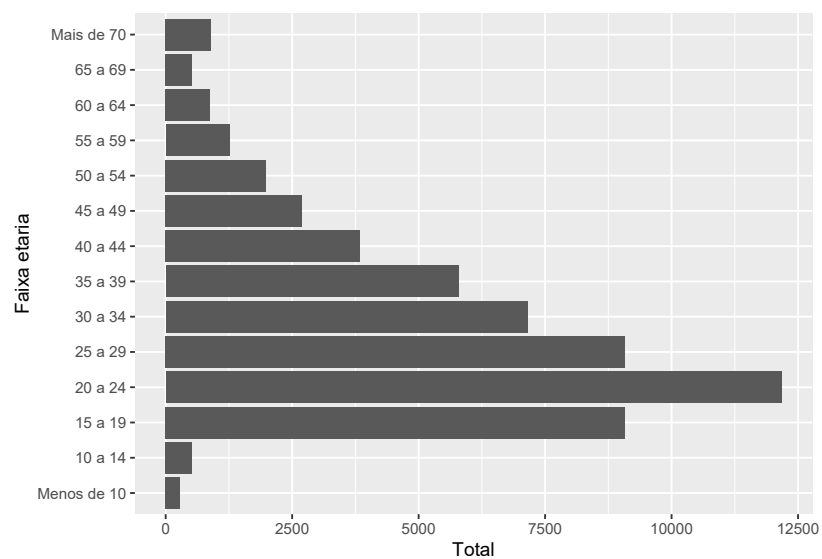
```
totais %>%
  ggplot() +
  geom_col(
    aes(y = `Faixa etaria`, x = Total)
  )
```

Portanto, a faixa de “Menos de 10” é a única faixa a ser reposicionada, e podemos realizar tal ação com a função `fct_relevel()`. Repare no exemplo abaixo, que após o ajuste, a faixa “Menos de 10” foi realocada para a posição mais inferior do eixo.

```
totais %>%
  mutate(
    `Faixa etaria` = fct_relevel(`Faixa etaria`, "Menos de 10")
  ) %>%
  ggplot() +
  geom_col(
    aes(y = `Faixa etaria`, x = Total)
  )
```



Fonte: Elaboração própria do autor.

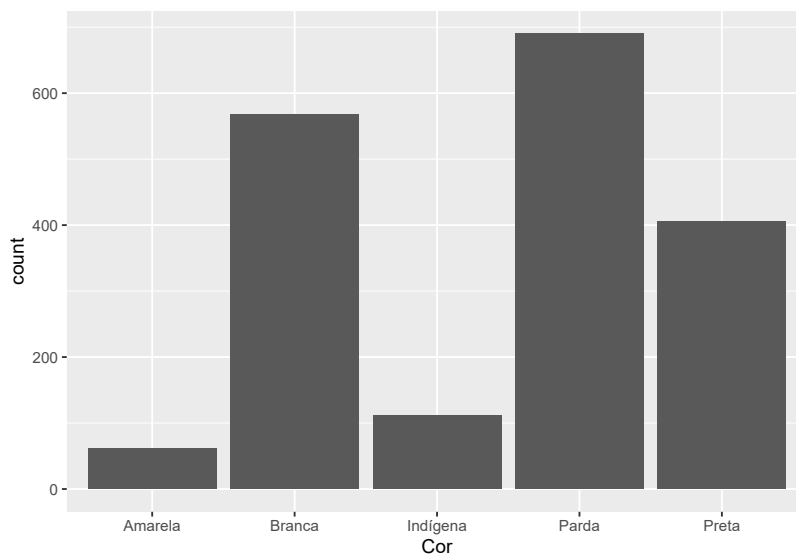


Fonte: Elaboração própria do autor.

Além dessas opções, a função `fct_infreq()` é muito útil para gráficos de barras do `ggplot`, que incluem por padrão um cálculo de frequência. Em outras palavras, ao lembrarmos que `fct_infreq()` busca reordenar um fator de acordo com a frequência em que os seus níveis aparecem em seus dados, se torna muito natural aliarmos essa função a um gráfico de barras do `ggplot`.

Por exemplo, se gerarmos um gráfico de barras a partir de cada cor de pele presente em nossa tabela `datasus`, temos o seguinte resultado:

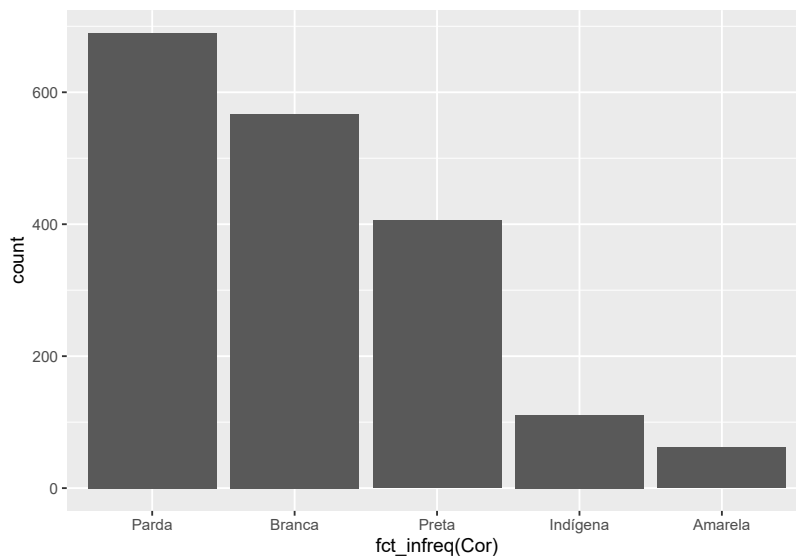
```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = Cor)  
  )
```



Fonte: Elaboração própria do autor.

Agora, com o uso de `fct_infreq()` podemos reposicionar essas barras em um sentido mais lógico, como está demonstrado abaixo:

```
datasus %>%  
  ggplot() +  
  geom_bar(  
    aes(x = fct_infreq(Cor))  
  )
```



Fonte: Elaboração própria do autor.

11.8 Modificando os níveis de um fator

Até o momento, demos bastante foco sobre a ordenação dos valores presentes no atributo `levels`. Justamente pelo fato de que essa característica define uma das principais vantagens do tipo `factor` no R, que é a de modificar a forma como a linguagem ordena os valores presentes em um vetor. Porém, ainda não discutimos o que ocorre quando nós deliberadamente alteramos um dos valores presentes no atributo `levels`.

Por exemplo, suponha que eu possua o fator abaixo. Nesse caso, o fator `f` possui quatro níveis, sendo eles: `a`, `b`, `c`, e `d`.

```
vec <- c("a", "c", "c", "d", "b", "a", "b")
f <- factor(vec, levels = c("a", "b", "c", "d"))
f
```

```
## [1] a c c d b a b
## Levels: a b c d
```

Agora, o que ocorre se eu tentar modificar o primeiro nível (`a`) desse fator? De maneira elegante e surpreendente, o R irá substituir todos os valores `a` presentes no fator, pelo novo valor definido, como está demonstrado abaixo:

```
levels(f) <- c("m", "b", "c", "d")
f
```

```
## [1] m c c d b m b
## Levels: m b c d
```

```

    "50 a 54", "55 a 59", "60 a 64"),
  "Acima de 64 anos" = c("65 a 69", "Mais de 70")
)
)

```

```
## # A tibble: 1,836 x 6
```

##	`Faixa etaria`	Genero	Cor	`Nome UF`	UF	Contagem
##	<fct>	<chr>	<chr>	<chr>	<chr>	<dbl>
##	1 Menos de 19 anos	Feminino	Parda	Acre	AC	4
##	2 Menos de 19 anos	Masculino	Parda	Acre	AC	4
##	3 Menos de 19 anos	Feminino	Branca	Acre	AC	2
##	4 Menos de 19 anos	Feminino	Parda	Acre	AC	4
##	5 Menos de 19 anos	Masculino	Branca	Acre	AC	6
##	6 Menos de 19 anos	Masculino	Parda	Acre	AC	65
##	7 Menos de 19 anos	Masculino	Preta	Acre	AC	1
##	8 Entre 20 e 64 anos	Feminino	Indígena	Acre	AC	1
##	9 Entre 20 e 64 anos	Feminino	Parda	Acre	AC	4
##	10 Entre 20 e 64 anos	Masculino	Branca	Acre	AC	7
##	# ... with 1,826 more rows					

Capítulo 12

Introdução à variáveis de tempo com lubridate

12.1 Introdução e pré-requisitos

Variáveis de tempo são aquelas que guardam **informações que se encontram em alguma unidade de tempo**. Exemplos são: datas (i.e. 20 de março de 2020), ou horários - que preferencialmente são acompanhados por uma data (i.e. 11:45 da manhã do dia 12 de fevereiro de 2001; ou, 12/02/2001 11:45:00), ou ainda, a duração (ou o tempo) de algum evento (12 segundos, 12 dias, 2 semanas, 1 mês e meio, etc.).

Tais variáveis podem ser interpretadas no R por meio de quatro tipos de dados diferentes, sendo eles: `Date`, `POSIXlt`, `POSIXct` e `difftime`. Logo, neste capítulo, vamos focar nesses quatro tipos de dados, e, introduzir várias ferramentas e operações comumente aplicadas sobre eles. Parte dessas ferramentas advém do pacote `lubridate` e, portanto, para acompanhar os exemplos deste capítulo, você deve (após instalar esse pacote em sua máquina) chamar por esse pacote em sua sessão, através do comando `library()`.

```
library(lubridate)
```

12.2 O pacote lubridate

Como é bem descrito por [Ripley e Hornik \(2001\)](#) e [Grothendieck e Petzoldt \(2004\)](#), desde sua versão 1.9, o R oferece “de fábrica” um excelente suporte para variáveis de tempo. Suas funções são capazes de lidar muito bem com diferenças entre fusos horários, além de incluírem anos bissextos e horários de verão. Porém, mesmo com esse potencial, essas funções (as `Date()`, as `POSIXct()`, `strptime()`, dentre outras) tendem a aplicar definições muito formais, tornando-as assim, pouca intuitivas para muitos usuários.

Por esse motivo, o pacote `lubridate` tem tido muito sucesso ao longo da comunidade, ao prover funções que realizam grande parte do trabalho irritante com essas funções. Ou seja, no fundo, várias das funções do pacote `lubridate` são apenas *wrappers*, isto é, são construídas a partir das funções

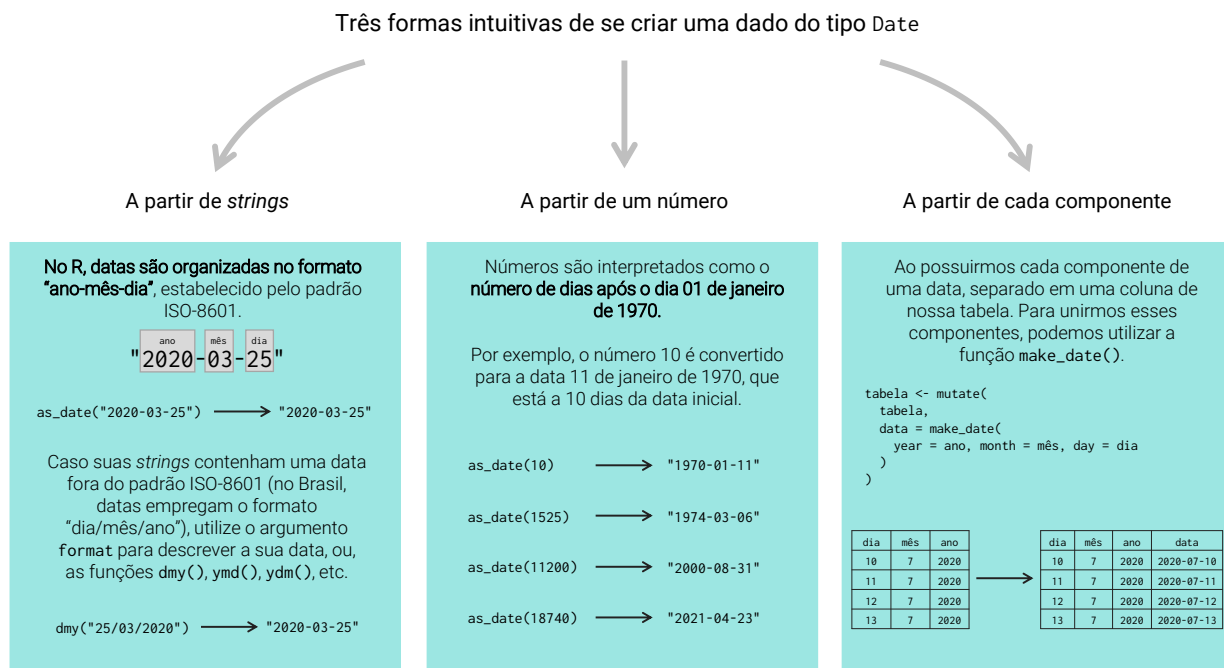
do pacote básico do R. Significa que o pacote `lubridate` foi criado, em grande parte, com o intuito de facilitar o nosso trabalho com as ferramentas que o R já oferece, ao invés de remodelá-las por completo.

Portanto, ao longo deste capítulo, você irá aprender primeiro sobre as funções do pacote `lubridate`, e, em seguida, as funções básicas do R são apresentadas para aqueles que desejam conhecer mais a fundo tal sistema. Dessa forma, nós estaremos apresentando primeiro, o atalho, e, em seguida, o caminho completo.

12.3 Datas com o tipo Date

No R, datas são normalmente interpretadas através do tipo de dado `Date`. Temos 3 métodos principais de se criar uma data no R (existem outros métodos menos intuitivos¹), os quais estão resumidos na figura 12.1 abaixo, e que são apresentados a seguir: 1) a partir de *strings* (um vetor do tipo `character`); 2) a partir de cada componente da data (dia, mês e ano); e 3) a partir de números.

Figura 12.1: Principais métodos para se criar datas no R



Fonte: Elaboração própria do autor.

Dentre as funções dos pacotes básicos do R, a função `as.Date()` é a principal função responsável

¹Em resumo, datas podem ser criadas a partir de todos os outros tipos que vimos até o momento (`character`, `double`, `integer`, `logical`, e `factor`). Para mais detalhes, consulte `?as.Date` e `?Date`.

por criar vetores do tipo `Date`. Todavia, ao longo dessa seção, estaremos focados nas funções do pacote `lubridate`, em especial, a função `as_date()`. De qualquer forma, saiba que, no fim das contas, as funções desse pacote vão utilizar a função `as.Date()` para criar o vetor contendo as suas datas. As funções `as_date()` e `as.Date()` são muito semelhantes entre si, logo, grande parte do conhecimento mostrado em `as_date()`, pode ser diretamente aplicado em `as.Date()`.

Ao longo das próximas seções, você pode rapidamente perceber que a formação de dados do tipo `Date` (assim como dos tipos `POSIXct` e `POSIXlt`) no R, envolve o ato de coerção de vetores que se encontram em outros tipos (como `character` ou `double`) para o tipo `Date`. Em outras palavras, não é possível criarmos diretamente um vetor do tipo `Date`. O motivo para tal prática, pode ser atribuído às diversas maneiras em que uma mesma data (além das outras variáveis de tempo) pode ser escrita, ou representada. Por essa diversidade, o R busca oferecer flexibilidade aos seus usuários, através de diferentes métodos de coerção. A figura 12.1, resume os principais métodos que vamos aprender ao longo dessa seção, além de algumas características importantes que envolvem o tipo `Date`.

12.3.1 A partir de *strings*

Como exemplo inicial, podemos fornecer à função `as_date()` (do pacote `lubridate`), a data 01 de Abril de 2020 como um *string*. Repare abaixo, que o resultado final da operação é um vetor do tipo `Date` (e não do tipo `character`).

```
d <- as_date("2020-04-01")
d
```

```
## [1] "2020-04-01"
```

```
class(d)
```

```
## [1] "Date"
```

Entretanto, você talvez tenha achado estranho o formato em que a data foi escrita na função `as_date()`. Pois no Brasil, datas são normalmente escritas no padrão “dia/mês/ano” (ex: 01/04/2020), e não “ano-mês-dia”. Este é o formato estipulado pelo padrão internacional ISO-8601, que é o padrão adotado pelo R. Ou seja, no R, datas são manipuladas, estocadas, fornecidas e apresentadas no formato “ano-mês-dia”.

Você irá rapidamente perceber que, muitos países podem escrever uma mesma data de maneiras muito diferentes. Por exemplo, nos EUA, datas são usualmente escritas no formato “mês-dia-ano” (ex: 02-18-2021), mas também aparecem muitas vezes em sua forma extensa (ex: February 18, 2021). Em algumas regiões da Espanha, datas são escritas no formato “ano/dia/mês” (ex: 2020/15/08). Também não é incomum encontrarmos em países nórdicos (Suécia, Finlândia, Dinamarca), datas escritas com o uso de pontos separando cada componente (ex: 2020.08.15).

Toda essa variedade só torna o nosso trabalho mais complicado, especialmente se a gente não sabe qual a origem, ou, o padrão adotado por essas datas. E não há nada que você possa fazer a respeito, a não ser, identificar o padrão adotado e ajustar a função empregada de acordo com esse padrão.

12.3.2 O que devo fazer se minhas datas se encontram em um formato diferente?

Portanto, caso você possua um conjunto de datas como *strings* (ou seja, em um vetor do tipo *character*), e, essas datas estejam em um formato diferente do estipulado pela ISO-8601, você tem 2 opções rápidas para transportar corretamente essas datas para o tipo *Date*.

Primeiro, todas as funções no R que lidam com variáveis de tempo, geralmente oferecem um argumento *format*, no qual você pode definir o formato, ou o padrão adotado por suas datas. Logo, você precisa apenas definir o argumento *format* em *as_date()*, ou em qualquer outra função que você esteja utilizando para essa coerção.

Segundo, você também pode utilizar as funções rápidas do pacote *lubridate*, *ymd()*, *dmy()*, *dym()* e *mdy()*, que já possuem uma ordem implícita, ou um *format* padrão. Dessa maneira, você economiza certo tempo, ao não ter que se preocupar com o argumento *format* nessas funções.

Por exemplo, suponha que você possua um conjunto de datas escritas no Brasil, guardadas no vetor *datas*, e que você deseja converter esse vetor (que se encontra no momento, no tipo *character*) para o tipo *Date*. Como os componentes da data estão na ordem “dia → mês → ano”, eu utilizo a função *dmy()* para ler essas datas.

```
datas <- c("15/03/2020", "16/03/2020", "18/03/2020", "24/03/2020")
dmy(datas)
```

```
## [1] "2020-03-15" "2020-03-16" "2020-03-18" "2020-03-24"
```

Isso significa que, a ordem na qual as letras “d”, “m” e “y” aparecem no nome da função, representa a ordem adotada pelo argumento *format* dessa função. Em outras palavras, a letra “d” simboliza o “dia”; a letra “m” por sua vez, o “mês”; e a letra “y”, o “ano”, ou, em inglês, “*year*”. Ou seja, a função *dmy()* espera como *input*, datas cujos componentes estejam na ordem “dia → mês → ano” (ou “d → m → y”). Já a função *ymd()*, tem como expectativa, datas cujos componentes estejam na ordem “ano → mês → dia” (ou “y → m → d”).

Portanto, as funções rápidas *dmy()*, *ymd()* e suas irmãs, possuem implicitamente uma ordem esperada para os componentes de suas datas. Para mais, essas funções identificam automaticamente qualquer caractere que não seja um dígito, e os trata como os delimitadores que separam cada componente da data. Logo, não importa se cada componente está sendo separado por um hífen (-), ponto (.), cifrão (\$) ou barra inclinada (/), essas funções serão capazes de detectar esses caracteres e ignorá-los durante a conversão.

Como já foi descrito acima, a segunda alternativa seria definirmos explicitamente o argumento *format* em *as_date()*. Neste argumento, você deve fornecer uma pequena definição² que descreve o padrão no qual a sua data se encontra. Para construir tal definição, você irá utilizar um conjunto

²A Universidade da Califórnia - Berkeley possui uma excelente página sobre como definir esses formatos, e que vale a pena conferir.

Tabela 12.1: Códigos que representam cada componente de uma data

Código	Significado	Exemplo
%d	Dia do mês (número)	24
%m	Mês (número)	12
%b	Mês (nome abreviado do mês)	Jan
%B	Mês (nome completo do mês)	January
%y	Ano (2 dígitos)	5
%Y	Ano (4 dígitos)	2005

Fonte: Elaboração própria do autor.

de códigos, que são formados pelo símbolo de porcentagem acompanhado de uma letra específica. Cada um desses códigos, podem representar um dos três componentes de uma data (dia, mês e ano). A tabela abaixo apresenta um resumo desses códigos.

Tendo os códigos acima, se uma data no Brasil é escrita no formato “dia/mês/ano”, uma descrição que representa tal padrão é "%d/%m/%Y". Como um outro exemplo, se temos as datas "2021, 30-12" e "97, 10, January", podemos utilizar respectivamente os valores "%Y, %d-%m" e "%y,%d,%B" para descrever os padrões adotados por cada uma. Veja os exemplos abaixo:

```
datas <- c("15/03/2020", "16/03/2020", "18/03/2020", "24/03/2020")
as_date(datas, format = "%d/%m/%Y")
```

```
## [1] "2020-03-15" "2020-03-16" "2020-03-18" "2020-03-24"
```

```
as_date("2021, 30-12", format = "%Y, %d-%m")
```

```
## [1] "2021-12-30"
```

```
as_date("97,10,January", format = "%y,%d,%B")
```

```
## [1] "1997-01-10"
```

```
as_date("12-30-1997", format = "%m-%d-%Y")
```

```
## [1] "1997-12-30"
```

Um detalhe importante é que os códigos %b e %B são capazes de representar apenas os nomes dos meses em inglês (ex: *April, December, October*). Por isso, se as suas datas possuem os nomes dos meses, em qualquer outra língua que não seja o inglês, você terá que, obrigatoriamente, traduzir esses nomes para o inglês, ou convertê-los para sua versão numérica (Março = 03; Abril = 04; Maio = 05; e assim por diante).

12.3.3 A partir de cada componente

Também é muito comum, termos cada um dos componentes separados em uma coluna específica de nossa tabela. Como exemplo, temos abaixo a tabela registros, onde o ano, mês e dia estão separados em uma determinada coluna da tabela.

Para unirmos esses componentes em uma data, nós podemos utilizar a função `make_date()`. Por meio dessa função, você precisa apenas conectar os argumentos `year`, `month` e `day`, aos nomes das colunas que contém o ano, mês e dia (respectivamente), de cada observação da tabela, como demonstrado abaixo.

```
registros <- tibble(
  valor = c(5.50, 4.25, 1.32, 24.10, 12.50),
  dia = c(5, 6, 8, 12, 15),
  mes = c(4, 4, 4, 4, 4),
  ano = c(2021, 2021, 2021, 2021, 2021)
)

registros <- mutate(
  registros,
  data = make_date(year = ano, month = mes, day = dia)
)

registros

## # A tibble: 5 x 5
##   valor   dia   mes   ano data
##   <dbl> <dbl> <dbl> <dbl> <date>
## 1  5.5     5     4  2021 2021-04-05
## 2  4.25    6     4  2021 2021-04-06
## 3  1.32    8     4  2021 2021-04-08
## 4 24.1   12     4  2021 2021-04-12
## 5 12.5   15     4  2021 2021-04-15
```

Além disso, é importante frisar que, os seus componentes não precisam necessariamente estar dentro de um `data.frame`. Dito de outra forma, você também pode fornecer cada componente de sua data como um vetor. Veja o exemplo abaixo:

```
dias <- c(1, 4, 12, 15, 7)
mes <- c(1, 1, 2, 2, 2)
ano <- c(2020, 2020, 2020, 2020, 2021)

make_date(year = ano, month = mes, day = dias)

## [1] "2020-01-01" "2020-01-04" "2020-02-12" "2020-02-15" "2021-02-07"
```

12.3.4 A partir de números

Para mais, o R também nos permite criar uma data a partir de um número. Por exemplo, eu posso criar a data "2020-04-01" (01 de Abril de 2020) através do número 18353. Repare abaixo, que ao invés de um vetor do tipo `double` contendo o número inicial (18353), a operação me retorna um vetor do tipo `Date`, contendo a data supracitada.

```
d <- as_date(18353)
d

## [1] "2020-04-01"

class(d)

## [1] "Date"
```

Quando você fornece um vetor numérico à função `as_date()`, todos os números contidos neste vetor são tratados como o número de dias desde a data "1970-01-01", ou, 01 de janeiro de 1970. Em outras palavras, o R utiliza uma “escala de dias”, e a data "1970-01-01" representa a origem, ou o ponto zero dessa escala (para representar dias anteriores a essa data, utilizamos números negativos). Nós denominamos essa data, como a **data de origem**.

Portanto, o número 18353 nos retorna a data "2020-04-01", pelo fato de que este dia está a 18353 dias de distância da data "1970-01-01". Caso você ainda sinta certa confusão, visite a seção **Como as variáveis de tempo são interpretadas pelo R ?**, que busca prover uma descrição mais formal e mais detalhada dos conceitos de data de origem e escala de tempo.

12.3.5 Fique atento aos tipos de dados empregados!

Vale a pena destacar que, apesar de serem apresentadas a você como *strings*, dados do tipo `Date` são guardados e interpretados de uma maneira completamente diferente dos dados do tipo `character`. Ou seja, quando valores do tipo `Date` aparecem em seu console, eles sempre aparecem contornados por aspas duplas, como se fossem dados do tipo `character`. **E não há qualquer informação aparente no console, que te indique qual dos dois tipos (`Date` ou `character`) está sendo empregado sobre esses valores.**

Por isso, é muito importante que você esteja atento à forma como o R está interpretando os seus dados. Use e abuse de funções e de testes lógicos que possam lhe assegurar que os seus dados estão sendo interpretados da maneira esperada! Tendo essas considerações em mente, a forma mais rápida de identificarmos se um vetor é do tipo `character`, ou do tipo `Date`, é descobriremos a sua classe, por meio da função `class()`. Repare no exemplo abaixo, que o primeiro valor pertence ao tipo `character`, enquanto o segundo, está sendo interpretado pelo tipo `Date`.

```
texto <- "2020-08-01"
data <- as.Date("2020-08-01")
class(texto)
```

```
## [1] "character"

class(data)

## [1] "Date"

### Um teste lógico para o tipo Date
class(texto) == "Date"

## [1] FALSE

class(data) == "Date"

## [1] TRUE
```

Por outro lado, caso as suas datas estejam dentro de um tibble, tal problemática perde um pouco de sua importância. Pois como descrevemos na seção **tibble's como uma alternativa moderna aos data.frame's**, quando um tibble aparece em seu console, ele sempre disponibiliza uma pequena descrição logo abaixo do nome de cada coluna, indicando o tipo de dado contido nela. Portanto, no exemplo abaixo, podemos rapidamente identificar pela descrição <date>, que os dados presentes na coluna data pertencem ao tipo de dado Date.

```
library(tibble)

tibble(
  data = dmy(c("20/05/2020", "21/05/2020", "22/05/2020", "23/05/2020"))
)

## # A tibble: 4 x 1
##   data
##   <date>
## 1 2020-05-20
## 2 2020-05-21
## 3 2020-05-22
## 4 2020-05-23
```

12.4 Datas, horários e fusos horários com POSIXct e POSIXlt

Em várias ocasiões, empresas, agentes e governos, precisam registrar o instante de ocorrência de algum episódio com um alto nível de precisão. Com isso, eu quero dizer que em certas situações, precisamos não apenas da data, mas também do horário e do fuso horário em que certo evento ocorre. Para isso, o R nos oferece os tipos POSIXct e POSIXlt, que são capazes de guardar não apenas datas, mas também horários além de fusos horários.

Figura 12.2: Como um ponto no tempo é definido nos tipos POSIXct e POSIXlt



Fonte: Elaboração própria do autor.

No fundo, o R utiliza as funções `as.POSIXct()` e `as.POSIXlt()` para criar um objeto dos tipos `POSIXct` e `POSIXlt`, respectivamente. Portanto, mesmo que as ferramentas apresentadas nessa seção pertençam (em sua maioria) ao pacote `lubridate`, saiba que no fundo, as funções `as.POSIXct()` e `as.POSIXlt()` são empregadas para criar o seu vetor do tipo `POSIXct` e `POSIXlt`.

Dentro da comunidade de R, vários usuários costumam se referir aos tipos `POSIXct` e `POSIXlt`, em uma forma mais intuitiva. Sendo o termo `date-time`, o mais utilizado para tal referência. Portanto, `date-time` é um sinônimo (ou uma gíria) utilizado para se referir à “espécie” de dado (isto é, uma data acompanhada de um horário e de um fuso horário) armazenado pelos tipos `POSIXct` e `POSIXlt`.

Por isso, ao longo dessa seção, quando estivermos descrevendo características gerais que se aplicam a ambos os tipos, vamos utilizar o termo `date-time` como um sinônimo aos tipos `POSIXct` e `POSIXlt`. Por outro lado, quando estivermos descrevendo características específicas de cada um, vamos utilizar o nome do tipo correspondente.

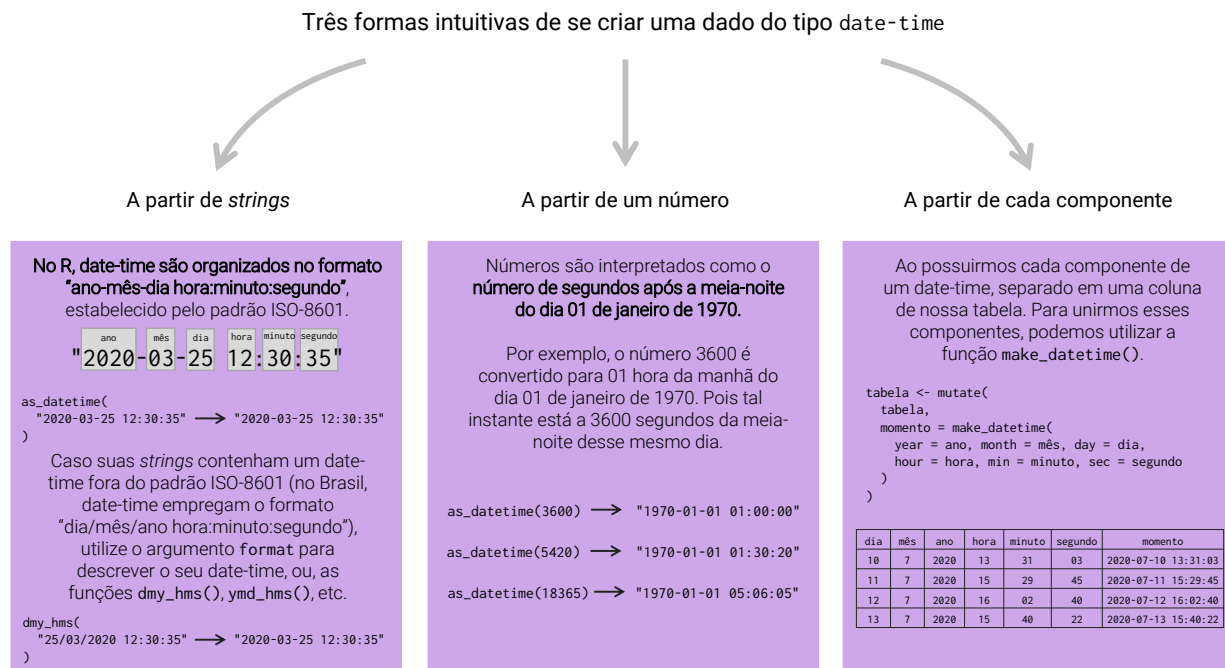
12.4.1 Criando vetores do tipo `date-time`

Para criarmos um vetor contendo dados do tipo `date-time`, podemos utilizar exatamente os mesmos métodos empregados no tipo `Date`, com pequenas modificações. Isto é, podemos criar um vetor dos tipos `POSIXct` e `POSIXlt`: 1) a partir de *strings*; 2) a partir de números; e 3) a partir de cada componente deste `date-time`. Um resumo de tais métodos, além de algumas observações quanto ao tipo `date-time`, são apresentados na figura 12.3.

Para realizar cada um desses métodos, o pacote `lubridate` nos oferece a função `as_datetime()`. Todavia, vale apontar que essa função sempre gera um vetor do tipo `POSIXct` como resultado. Por isso, se você deseja converter o seu objeto para o tipo `POSIXlt`, aplique a função `as.POSIXlt()`

sobre o resultado de `as_datetime()`.

Figura 12.3: Principais métodos para se criar dados do tipo date-time no R



Fonte: Elaboração própria do autor.

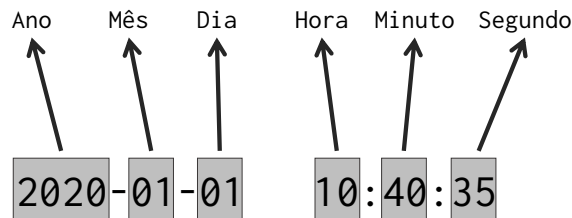
12.4.2 A partir de strings

Assim como descrevemos durante as seções do tipo Date, o R segue as regras definidas no padrão internacional ISO-8601 para armazenar e interpretar suas variáveis de tempo. Esse padrão delimita que, dados do tipo date-time devem ser escritos no formato “ano-mês-dia hora:minuto:segundo”. A figura 12.4, provê uma representação visual de tal formato.

Mais abaixo, temos um exemplo em que um objeto chamado `dt` é criado, com o objetivo de guardar o seguinte ponto no tempo: 10 horas, 40 minutos e 35 segundos do dia 01 de janeiro de 2020. Repare nesse exemplo, que nós não incluímos no *string* inicial qualquer informação a respeito do fuso horário utilizado. Mesmo assim, a função `as_datetime()` automaticamente configurou o horário com o fuso UTC, que corresponde à *Coordinated Universal Time*. Portanto, sempre que você não definir explicitamente um fuso horário, a função `as_datetime()` vai utilizar o fuso horário UTC.

Para mais, isso demonstra que **não é necessário incluirmos o fuso horário utilizado, diretamente no *string* a ser fornecido**. Pois tal informação é definida separadamente no argumento `tz` da função.

Figura 12.4: Formato padrão no R para dados do tipo date-time



Fonte: Elaboração própria do autor.

```
dt <- as_datetime("2020-01-01 10:40:35")
dt
```

```
## [1] "2020-01-01 10:40:35 UTC"
```

Isso não significa que, os *strings* não devem ou não podem conter qualquer informação a respeito do fuso horário. Mas significa que essas informações serão, por padrão, ignoradas pela função, que vai utilizar o fuso UTC para qualquer *input*. Veja o exemplo abaixo, em que dois fusos horários diferentes são testados, e o mesmo resultado é gerado em ambos.

```
as_datetime("2020-01-01 10:40:35 Portugal")
```

```
## [1] "2020-01-01 10:40:35 UTC"
```

```
as_datetime("2020-01-01 10:40:35 America/Sao_Paulo")
```

```
## [1] "2020-01-01 10:40:35 UTC"
```

Portanto, a maneira correta de definir o fuso horário a ser empregado, é por meio do argumento `tz`, como demonstrado abaixo:

```
as_datetime("2020-01-01 10:40:35", tz = "Portugal")
```

```
## [1] "2020-01-01 10:40:35 WET"
```

```
as_datetime("2020-01-01 10:40:35", tz = "America/Sao_Paulo")
```

```
## [1] "2020-01-01 10:40:35 -03"
```

Fusos horários são usualmente fornecidos ao argumento `tz` por meio de um código (e.g. "WET", "UTC", "ROK", "CET", etc.), ou por meio de uma referência de região ou cidade específica (e.g. "Europe/Paris", "Pacific/Auckland", "America/Toronto", etc.). Para consultar a lista completa de valores reconhecidos pelo R, execute a função `OlsonNames()` em seu console.

Dito de outra forma, valores como "ROK" (abreviação para *Republic of Korea*), "CET" (*Central European Time*), "America/Sao_Paulo" (cidade de São Paulo) e "Pacific/Auckland" (cidade de Auckland), são aceitos pelo argumento `tz`, porque eles estão inclusos no resultado da função `OlsonNames()`. Em contraste, valores como "São Paulo", "WST", e "+11", não são aceitos pelo argumento `tz`, pois não estão presentes em `OlsonNames()`.

```
as_datetime("2020-01-01 10:34:12", tz = "CET")
```

```
## [1] "2020-01-01 10:34:12 CET"
```

```
as_datetime("2020-01-01 10:34:12", tz = "ROK")
```

```
## [1] "2020-01-01 10:34:12 KST"
```

```
as_datetime("2020-01-01 10:34:12", tz = "Pacific/Auckland")
```

```
## [1] "2020-01-01 10:34:12 NZDT"
```

```
### Quando incluímos um fuso horário desconhecido
```

```
### pelo R, a seguinte mensagem de erro aparece:
```

```
as_datetime("2020-01-01 10:34:12", tz = "WST")
```

```
Error in C_force_tz(time, tz = tzone, roll) :
```

```
  CCTZ: Unrecognized output timezone: "WST"
```

12.4.3 O que devo fazer se meus dados se encontram em um formato diferente?

Da mesma maneira que uma mesma data pode ser escrita de várias formas, horários também podem assumir formatos diferentes. Sendo que, diferentemente das datas, algumas partes de um horário (hora, minuto e segundo) podem ser ignoradas, a depender da precisão de tempo necessária. De qualquer modo, em casos como este, as soluções a serem empregadas são exatamente as mesmas que descrevemos na seção [O que devo fazer se minhas datas se encontram em um formato diferente?](#), que são:

- 1) Utilizar os códigos oferecidos pelo R no argumento `format` da função.
- 2) Ou utilizar as funções rápidas do pacote `lubridate` (`dmy_h()`, `dmy_hm()`, `dmy_hms()`, etc.) que possuem uma ordem implícita para cada componente.

A tabela abaixo, apresenta os principais códigos oferecidos pelo R para cada componente de um dado do tipo `date-time`. Porém, há vários outros códigos, os quais são menos comuns e, que por isso, foram omitidos da tabela abaixo. Você pode encontrar uma lista completa desses códigos, ao consultar a documentação interna da função `strptime()`, com o comando `?strptime`.

Pelas informações dispostas na tabela abaixo, sabemos que o formato delineado pelo padrão ISO-8601, isto é, “ano-mês-dia hora:minuto:segundo”, pode ser descrito pelo padrão `"%Y-%m-%d %H:%M:%S"`, ou, de forma mais sucinta, `"%F %T"`. Como exemplo, repare abaixo que ambas as descrições geram os mesmos resultados, quando aplicadas sobre os valores presentes no vetor `pontos`:

```
pontos <- c("2018-06-15 08:11:05", "2018-07-22 21:09:05")
as_datetime(pontos, format = "%Y-%m-%d %H:%M:%S")

## [1] "2018-06-15 08:11:05 UTC" "2018-07-22 21:09:05 UTC"

### Ou de forma análoga
as_datetime(pontos, format = "%F %T")

## [1] "2018-06-15 08:11:05 UTC" "2018-07-22 21:09:05 UTC"
```

No caso do Brasil, valores do tipo `date-time` costumam se apresentar no formato “dia/mês/ano hora:minuto:segundo”. Logo, uma descrição capaz de representar tal formato é `"%d/%m/%Y %H:%M:%S"`, ou então, uma alternativa mais curta é `"%d/%m/%Y %T"`.

```
pontos_br <- c("15/06/2018 08:11:05", "22/07/2018 21:09:05")
as_datetime(pontos_br, format = "%d/%m/%Y %H:%M:%S")

## [1] "2018-06-15 08:11:05 UTC" "2018-07-22 21:09:05 UTC"

### Ou de forma análoga
as_datetime(pontos_br, format = "%d/%m/%Y %T")

## [1] "2018-06-15 08:11:05 UTC" "2018-07-22 21:09:05 UTC"
```

Vale ressaltar que, em todos os exemplos mostrados até agora, todos os componentes de um `date-time` foram fornecidos nos *strings* utilizados como *input*. Dito de outra forma, em nenhum exemplo mostrado até o momento, os segundos, os minutos ou as horas estavam ausentes dos *strings* utilizados como *input*. Esse cenário perfeito nem sempre ocorre, e isso não necessariamente é um problema sério. Pois, em alguns processos, a empresa nem sempre precisa de uma precisão muito alta em seus registros de tempo.

Por exemplo, uma indústria de aço não recebe matérias primas a cada segundo. Muitas vezes, a firma encomenda um grande estoque de materiais, combustíveis e minérios ao final de cada mês.

Tabela 12.2: Códigos que podem representar cada componente de um dado do tipo date-time

Código	Significado	Exemplo de valores que o código representa
Componentes de datas		
%d	Dia do mês (número).	24
%m	Mês (número).	12
%b	Mês (nome abreviado do mês).	Jan
%B	Mês (nome completo do mês).	January
%y	Ano (2 dígitos).	05
%Y	Ano (4 dígitos).	2005
%j	Dia do ano (número de 0 a 366). O número 366 só é permitido em anos bissextos.	127
%F	Equivalente a %Y-%m-%d (que é o formato da ISO-8601 para datas).	2020-01-15
Componentes de horários		
%H	Hora (número de 0 a 23).	15 (isto é, 3 horas da tarde)
%I	Hora (número de 1 a 12). Nesse caso, a descrição deve também incluir o código %p, que indica se a hora é da manhã ou da tarde.	11 (dependendo do valor de %p, pode ser 11 da tarde ou 11 da manhã)
%M	Minuto (número de 0 a 59).	52
%p	Índice que indica manhã e tarde (AM/PM).	AM
%S	Segundos (número de 0 a 61). Intervalo vai até os 61 como forma de permitir os dois segundos a mais presentes em anos bissextos.	59
%T	Equivalente à %H:%M:%S.	14:25:59
Componentes da semana		
%u	Dia da semana (número de 1 a 7). Número 1 representa segunda-feira.	3 (que representa quarta-feira)
%a	Nome abreviado do dia semana.	Mon (que representa Monday, ou segunda-feira em inglês)
%A	Nome completo do dia semana.	Monday (que representa segunda-feira em inglês)
Componentes do fuso-horário		
%z	Um ajuste, ou uma compensação (em horas e minutos) a ser adicionada ao fuso horário aplicado sobre o resultado da função	-0300

Fonte: Elaboração própria do autor.

Por esse motivo, a firma talvez precise registrar apenas as horas e minutos do dia, em que cada entrega (ou carregamento de matéria-prima) chegou a sua sede, em um determinado mês.

Tendo isso em mente, se eu possuo o *string* abaixo, contendo o valor "2020-04-15 10:30", eu poderia utilizar a descrição "%F %H:%M" para descrever o formato em que esse valor se encontra. Contudo, uma alternativa eficiente e intuitiva, é utilizar a função `ymd_hm()`. Perceba pelo resultado abaixo, que ao não possuímos um determinado componente de um dado do tipo *date-time*, esse componente faltante é preenchido por zeros.

```
dt <- "2020-04-15 10:30"
as_datetime(dt, format = "%F %H:%M")
```

```
## [1] "2020-04-15 10:30:00 UTC"
```

```
ymd_hm(dt)
```

```
## [1] "2020-04-15 10:30:00 UTC"
```

Do mesmo modo que descrevemos anteriormente, funções rápidas como `ymd_hm()` possuem uma ordem para cada componente que está implícita no nome dessa função. A novidade em relação às funções `ymd()`, `dmy()` e suas irmãs, é que essas funções focadas em dados do tipo *date-time*, incluem três novas letras que se referem a hora (h), minuto (m) e segundo (s). Portanto, sabemos pelo nome da função, que `ymd_hm()` espera um *input* onde os componentes se apresentam na ordem “ano → mês → dia → hora → minuto”.

Por isso, a função `dmy_hms()` é uma alternativa ideal para ler dados do tipo *date-time* que foram escritos segundo o padrão brasileiro. Pois essa função espera como *input*, um dado em que os componentes seguem a ordem “dia → mês → ano → hora → minuto → segundo”. Veja o exemplo abaixo:

```
dts <- c("12/10/1998 19:19:32", "12/10/1998 22:15:09")
sem_segundo <- c("12/10/1998 19:19", "12/10/1998 22:15")
sem_minuto_e_segundo <- c("12/10/1998 19", "12/10/1998 22")
```

```
dmy_hms(dts)
```

```
## [1] "1998-10-12 19:19:32 UTC" "1998-10-12 22:15:09 UTC"
```

```
dmy_hm(sem_segundo)
```

```
## [1] "1998-10-12 19:19:00 UTC" "1998-10-12 22:15:00 UTC"
```

```
dmy_h(sem_minuto_e_segundo)
```

```
## [1] "1998-10-12 19:00:00 UTC" "1998-10-12 22:00:00 UTC"
```

Para além dos códigos mais tradicionais, a parte inferior da tabela x.x descreve alguns códigos menos comuns, como o código %z. Esse código em específico, é capaz de identificar um ajuste (em horas e minutos) presente no *string* de *input*, e adicioná-lo ao fuso horário aplicado sobre o resultado da função. Porém, como veremos mais a frente, lidar com fusos horários não é algo muito simples, e para piorar, o cálculo aritmético por trás da adição de um ajuste com o código %z, é no mínimo, peculiar.

Portanto, o uso de ajustes representados pelo código %z, é algo mais complicado do que uma simples adição³. Sendo que, o cálculo aritmético aplicado por tal código, está demonstrado em detalhes na seção **Quando fusos horários diferentes geram uma grande confusão**. Por esses motivos, o código %z é algo difícil de se trabalhar, especialmente levando-se em conta que: **em certas situações, o código %z gera resultados diferentes entre as funções `as_datetime()` e `as.POSIXct()`**.

Tal diferença, se baseia puramente no fato de que a função `as_datetime()` escolhe, por padrão, o fuso horário UTC, enquanto a função `as.POSIXct()`, tende a escolher o fuso horário padrão de seu sistema operacional (o qual não necessariamente é o fuso horário UTC). Como exemplo, temos abaixo um vetor chamado `dt`, que contém o instante: 8 horas do dia 01 de janeiro de 2020. Ademais, podemos identificar um ajuste negativo (ou “para trás”) de 3 horas (-0300), ao final do *string*. Perceba abaixo, que ambas as funções nos retornam horários diferentes. Esse problema vale certa reflexão sua, porque essa diferença existe? Como ela ocorre? Vamos dar as respostas para essas perguntas mais a frente. Até lá, pense um pouco sobre esses resultados.

```
dt <- "2020-01-01 08:00:00 -0300"  
strptime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 08:00:00"
```

```
as_datetime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 11:00:00 UTC"
```

Como uma dica, repare como os resultados mudam quando adicionamos uma hora ao ajuste, gerando assim, um desvio negativo de 2 horas (-0200).

```
dt <- "2020-01-01 08:00:00 -0200"  
strptime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 07:00:00"
```

³Se é algo não muito comum, e ainda, complicado de se entender, então porque falar sobre o código %z? Pelo simples fato de que ele demonstra de forma eficiente, como o uso de fusos horários podem contribuir para a confusão de muitos usuários.

```
as_datetime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 10:00:00 UTC"
```

12.4.4 A partir de cada componente

Caso você possua, separadamente, cada um dos itens que compõe um dado do tipo date-time (dia, mês, ano, hora, minuto e segundo), você pode utilizar a função `make_datetime()` para uní-los em um único vetor do tipo date-time. Por exemplo, suponha que você possua um `data.frame` parecido com a tabela `tab` abaixo.

```
tab <- tibble(
  ano = 2020,
  mês = 5,
  dia = c(15, 16, 16, 18, 19),
  hora = c(9, 11, 12, 8, 14),
  minuto = c(7, 23, 19, 15, 30),
  segundo = c(34, 11, 5, 17, 49)
)
```

tab

```
## # A tibble: 5 x 6
##   ano  mês  dia  hora minuto segundo
##   <dbl> <dbl> <dbl> <dbl>   <dbl>   <dbl>
## 1  2020     5   15     9       7       34
## 2  2020     5   16    11      23       11
## 3  2020     5   16    12      19        5
## 4  2020     5   18     8      15       17
## 5  2020     5   19    14      30       49
```

Em relação à função `make_date()`, a função `make_datetime()` introduz três novos argumentos, sendo eles: `hour`, `min` e `sec`, que se referem às horas, os minutos e os segundos, respectivamente.

```
tab <- mutate(
  tab,
  date_time = make_datetime(
    year = ano, month = mês, day = dia,
    hour = hora, min = minuto, sec = segundo
  )
)
```

tab

```
## # A tibble: 5 x 7
##   ano   mês   dia  hora minuto segundo date_time
##   <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dtm>
## 1  2020     5    15     9      7      34 2020-05-15 09:07:34
## 2  2020     5    16    11     23     11 2020-05-16 11:23:11
## 3  2020     5    16    12     19      5 2020-05-16 12:19:05
## 4  2020     5    18     8     15     17 2020-05-18 08:15:17
## 5  2020     5    19    14     30     49 2020-05-19 14:30:49
```

12.4.5 A partir de números

Assim como ocorre no tipo `Date`, dados do tipo `date-time` também podem ser criados a partir de números. O mecanismo de conversão é muito semelhante ao que mostramos com o tipo `Date`. Porém, ao invés de representar o número de dias desde uma data de origem, ao ser convertido para os tipos `POSIXct` e `POSIXlt`, o número que estamos tentando converter, será interpretado como o número de segundos desde a meia-noite de 01 de janeiro de 1970 (de outra forma, 1970-01-01 00:00:00).

Por isso, podemos chegar ao instante “08 horas do dia 01 de janeiro de 2005”, ao convertermos o número 1.104.566.400 (que representa aproximadamente 1,104 bilhões de segundos) para o tipo `date-time`. Em outras palavras, 08 horas do dia 01 de janeiro de 2005 está a 1.104.566.400 segundos de distância da meia-noite do dia 01 de janeiro de 1970.

```
as_datetime(1104566400)
```

```
## [1] "2005-01-01 08:00:00 UTC"
```

Para mais detalhes sobre os conceitos de ponto de origem e escala temporal (que são essenciais para se compreender corretamente essa conversão entre números e instantes no tempo), consulte a seção [Como as variáveis de tempo são interpretadas pelo R ?](#).

12.4.6 Novamente, fique atento aos tipos empregados!

Assim como ocorre com o tipo `Date`, dados do tipo `POSIXct` e `POSIXlt` também aparecem em seu console, contornados por aspas duplas e, novamente, não há qualquer informação aparente, que nos informe se os dados em questão encontram-se no tipo `character` ou em algum dos tipos `date-time`. Da mesma forma que descrevemos no tipo `Date`, uma maneira simples e prática de identificar se um objeto pertence ao tipo `POSIXct` ou `POSIXlt`, é olhar para a classe desse objeto, com a função `class()`. Entretanto, de maneira diferente do tipo `Date`, que continha apenas um valor para a sua classe, repare abaixo que, objetos dos tipos `POSIXct` e `POSIXlt` sempre possuem um segundo valor para a sua classe (`POSIXt`).

```
v_POSIXct <- as.POSIXct("2020-01-01 10:40:35")
v_POSIXlt <- as.POSIXlt("2020-01-01 10:40:35")
```



```
class(v_POSIXct)
```

```
## [1] "POSIXct" "POSIXt"
```

```
class(v_POSIXlt)
```

```
## [1] "POSIXlt" "POSIXt"
```

Por esse detalhe, quando você estiver criando o seu teste lógico, utilize o operador `%in%`, ao invés do operador `==`.

```
### Para identificar se o objeto é do
### tipo POSIXct, utilize:
"POSIXct" %in% class(v_POSIXct)
```

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

```
### Já para o tipo POSIXlt, utilize:
"POSIXlt" %in% class(v_POSIXlt)
```

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

Um outro método útil de identificarmos se um objeto pertence aos tipos `POSIXct` e `POSIXlt`, é através da função `inherits()`, como está demonstrado abaixo:

```
inherits(v_POSIXct, "POSIXct")
```

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

```
inherits(v_POSIXlt, "POSIXlt")
```

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

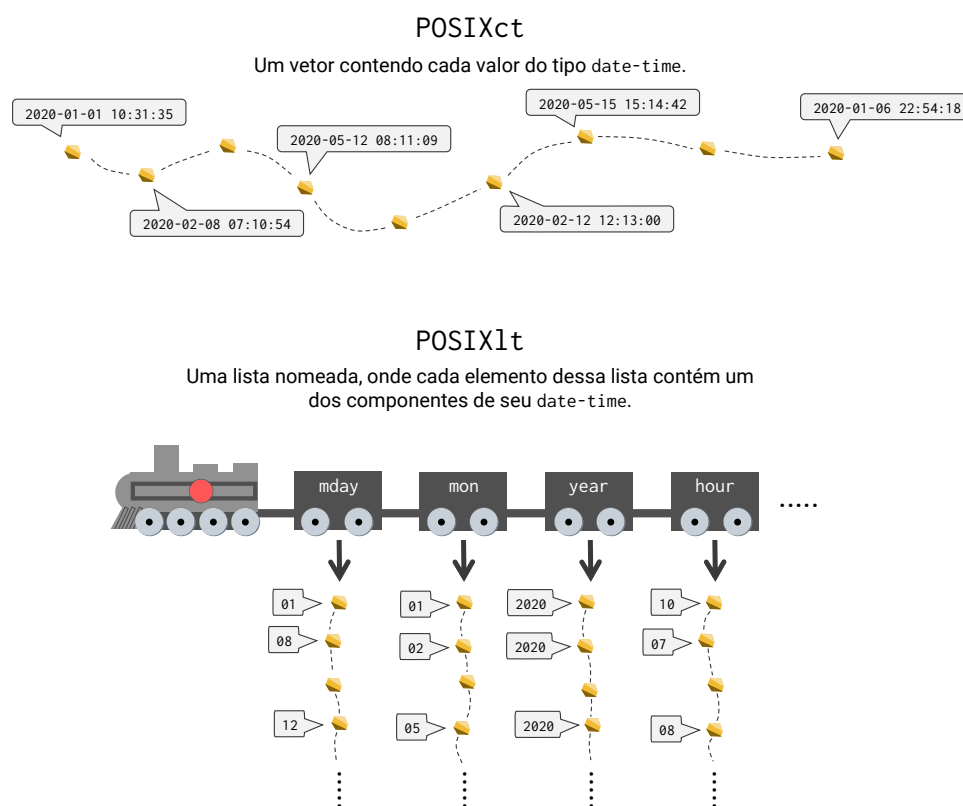
12.5 Diferenças entre POSIXct e POSIXlt

Até o momento, nós não descrevemos quais são as características que diferem os tipos `POSIXct` e `POSIXlt` um do outro. Em resumo, os valores do tipo `POSIXct` são guardados dentro de um vetor e, os valores do tipo `POSIXlt`, em uma lista nomeada contendo vários vetores que guardam cada componente desses valores.

Em mais detalhes, quando utilizamos o tipo `POSIXct`, o R vai apenas construir um vetor contendo os nossos dados do tipo date-time. Apesar desses valores serem apresentados a nós, no formato

“ano-mês-dia hora:minuto:segundo”, em uma camada mais profunda, o R vai armazená-los como o número de segundos desde o instante 1970-01-01 00:00:00 (meia-noite de 01 de janeiro de 1970). Por outro lado, quando empregamos o tipo POSIXlt, o R vai construir uma lista nomeada contendo vários vetores, onde cada um desses vetores possui um componente específico (dia, mês, ano, hora, etc.) de seu dado do tipo date-time. A figura 12.5, fornece uma representação visual dessa diferença.

Figura 12.5: Representação visual das estruturas formadas pelos tipos POSIXct e POSIXlt



Fonte: Elaboração própria do autor.

Para mais detalhes, você pode conferir a documentação interna desses tipos, com `?POSIXct` ou `?POSIXlt`. Para mais, vale destacar que, dados que se encontram em qualquer um desses dois tipos, são apresentados da mesma maneira a nós. Em outras palavras, quando aparecem em seu console do R, os dados do tipo POSIXct e POSIXlt sempre aparecem como um vetor contendo valores no formato “ano-mês-dia hora:minuto:segundo”.

```
v_POSIXct <- as.POSIXct("2020-01-01 10:40:35")
v_POSIXlt <- as.POSIXlt("2020-01-01 10:40:35")
```

```
print(v_POSIXct)
```

```
## [1] "2020-01-01 10:40:35 -03"
```

```
print(v_POSIXlt)
```

```
## [1] "2020-01-01 10:40:35 -03"
```

Porém, como eu descrevi acima, a forma como esses dados estão estruturados dentro do objeto é completamente diferente. Por exemplo, eu posso extrair os segundos (35) do valor (ou valores) em questão, ao acessar o item de nome `sec` da lista que compõe o objeto `v_POSIXlt`. Da mesma forma, caso eu precise extrair o dia (01) de cada data presente no objeto `v_POSIXlt`, basta acessar o item de nome `mday` dessa lista.

```
v_POSIXlt$sec
```

```
## [1] 35
```

```
v_POSIXlt$mday
```

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

Na hipótese de, realizarmos a mesma tarefa com um valor do tipo `POSIXct`, sem que ele seja convertido para o tipo `POSIXlt`, a nossa melhor opção seria implantarmos um *string subsetting*, com as funções que já vimos no capítulo [Manipulação e transformação de strings com stringr](#).

```
library(stringr)
```

```
como_texto <- as.character(v_POSIXct)
```

```
### Por exemplo, para extrair os segundos faríamos:
```

```
as.integer(str_sub(como_texto, 18, 19))
```

```
## [1] 35
```

```
### Para extrair o dia:
```

```
as.integer(str_sub(como_texto, 9, 10))
```

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

```
### Para extrair o ano:
```

```
as.integer(str_sub(como_texto, 1, 4))
```

```
## [1] 2020
```

Portanto, podemos dizer que o tipo `POSIXlt` provê um formato mais próximo da maneira como nós, seres humanos, pensamos sobre um valor do tipo `date-time`. Ou seja, diversos componentes (dia, mês, ano, hora, etc.) que, em conjunto, formam um dado do tipo `date-time`, mas que ainda representam unidades ou informações diferentes. Todavia, o tipo `POSIXct` fornece uma estrutura muito mais ideal para o uso em `data.frame`'s e, em geral, operações e cálculos aritméticos. Dito de outra forma, seja em uma coluna de um `data.frame` ou em qualquer outra estrutura, é mais fácil trabalhar no R com dados do tipo `date-time`, quando eles se encontram no tipo `POSIXct`.

Você também pode encarar a escolha entre esses dois tipos da seguinte maneira: se você deseja extrair um componente específico de cada data (dia, mês, ano, hora, etc.), você pode transformar os seus valores para o tipo `POSIXlt`, com a função `as.POSIXlt()` e, em seguida, extrair o item da lista resultante que contém o componente desejado; caso você não tenha pretensões de extrair algum componente, sempre utilize o tipo `POSIXct`. Pois esse tipo provê um formato mais natural para diversas operações e cálculos aritméticos que você venha a realizar sobre os seus valores.

12.6 Extrair os componentes de uma variável de tempo

Você já sabe que, nós podemos extrair cada componente de maneira simples e intuitiva, ao transformarmos os dados em questão para o tipo `POSIXlt`, e utilizar os itens da lista resultante para chegarmos ao componente desejado. Porém, também vamos mostrar nessa seção, algumas funções rápidas presentes no pacote `lubridate`, que tornam esse processo de extração ainda mais simples.

Primeiro, essas funções rápidas e as partes extraídas por cada uma delas, são:

- `day()`, dia do mês (1-31).
- `month()`, mês do ano (1-12).
- `year()`, ano (número de 4 dígitos).
- `hour()`, hora do dia (0-23).
- `minute()`, minutos (0-59).
- `second()`, segundos (0-61).

Tendo essas funções em mente, se nós desejamos extrair apenas as horas de cada valor presente no vetor `dt` abaixo, nós podemos simplesmente aplicar a função `hour()` sobre este vetor. De modo análogo, se desejamos calcular o dia do mês correspondente a cada valor, nós podemos utilizar a função `day()`:

```
dt <- c("21/02/2020 10:22:53", "01/11/2019 20:13:01", "19/07/2018 15:24:20")
dt <- dmy_hms(dt)
```

```
hour(dt)
```

```
## [1] 10 20 15
```

```
minute(dt)
```

```
## [1] 22 13 24
```

```
day(dt)
```

```
## [1] 21 1 19
```

Essas funções rápidas são particularmente úteis, quando desejamos extrair os componentes de de alguma coluna de um `data.frame`'s. Como exemplo, podemos visitar novamente a tabela `transf` que vimos ao longo do capítulo 4, e extrair os componentes de cada valor presente em sua coluna `Data`.

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

```
library(readr)
transf <- read_csv2(paste0(github, pasta, arquivo))
```

```
transf <- transf %>%
  select(-Descricao) %>%
  mutate(
    hora = hour(Data),
    minuto = minute(Data),
    segundo = second(Data)
  )
```

```
transf
```

```
## # A tibble: 20,006 x 8
```

##	Data	Usuario	Valor	TransferID	Pais	hora	minuto	segundo
##	<dtm>	<chr>	<dbl>	<dbl>	<chr>	<int>	<int>	<dbl>
## 1	2018-12-06 22:19:19	Eduardo	599.	116241629	Alem~	22	19	19
## 2	2018-12-06 22:10:34	Júlio	4611.	115586504	Alem~	22	10	34
## 3	2018-12-06 21:59:50	Nathál~	4418.	115079280	Alem~	21	59	50
## 4	2018-12-06 21:54:13	Júlio	2740.	114972398	Alem~	21	54	13
## 5	2018-12-06 21:41:27	Ana	1408.	116262934	Alem~	21	41	27
## 6	2018-12-06 21:18:40	Nathál~	5052.	115710402	Alem~	21	18	40
## 7	2018-12-06 20:54:32	Eduardo	5665.	114830203	Alem~	20	54	32
## 8	2018-12-06 20:15:46	Sandra	1474.	116323455	Alem~	20	15	46
## 9	2018-12-06 20:04:35	Armando	8906.	115304382	Alem~	20	4	35
## 10	2018-12-22 20:00:56	Armando	18521.	114513684	Alem~	20	0	56

```
## # ... with 19,996 more rows
```

Para mais, o vetor `dt`, assim como a coluna `Data` da tabela `transf`, que utilizamos nos exemplos anteriores, são vetores do tipo `POSIXct`. Contudo, as funções mostradas acima, funcionam exatamente da mesma forma com valores do tipo `Date`. Ou seja, o processo é o mesmo, basta aplicar a função que extrai o componente no qual você está interessado sobre o seu vetor do tipo `Date`.

```
v_Date <- c("21/02/2020", "01/11/2019", "19/07/2018")
v_Date <- dmy(v_Date)
```

```
day(v_Date)
```

```
## [1] 21  1 19
```

```
month(v_Date)
```

```
## [1]  2 11  7
```

```
year(v_Date)
```

```
## [1] 2020 2019 2018
```

Em outras palavras, isso é a mesma coisa que dizer: “um vetor do tipo `Date` pode ser convertido diretamente para o tipo `POSIXlt` e, com isso, podemos extrair os componentes que compõe cada data presente nesse vetor”. Tal fato está exposto no exemplo abaixo:

```
v_Date <- c("21/02/2020", "01/11/2019", "19/07/2018")
v_Date <- as.POSIXlt(dmy(v_Date))
```

```
v_Date$yday
```

```
## [1] 21  1 19
```

Como definimos na seção anterior, um objeto do tipo `POSIXlt` é na realidade, uma lista nomeada, e, você pode descobrir quais são os nomes de cada item dessa lista, ao acessar o atributo `names` desse objeto, com a função `attr()`, como demonstrado abaixo. Dessa maneira, o nome `"hour"` me indica que há um item chamado `hour` nessa lista, e, ao acessar esse item com o comando `v_POSIXlt$hour`, eu posso identificar o que se encontra dentro desse item. Você pode encontrar mais detalhes sobre cada item dessa lista, ao ler a documentação interna do tipo `POSIXlt`, com o comando `?POSIXlt`.

```
attr(v_POSIXlt, "names")
```

```
## [1] "sec"    "min"    "hour"   "yday"   "mon"    "year"   "yday"
## [8] "yday"   "isdst"  "zone"   "gmtoff"
```

12.7 Fusos horários

12.7.1 Como identificar o fuso horário associado a um valor do tipo date-time

É importante destacar que, todo dado do tipo `POSIXct` ou `POSIXlt` estará sempre ligado a algum fuso-horário de referência, mesmo que esse fuso não esteja evidente a primeira vista. Há duas maneiras principais de se identificar o fuso utilizado: primeiro, veja se alguma informação aparece ao lado do horário presente em seu valor do tipo date-time; segundo, veja a informação armazenada no atributo `tzzone` de seu objeto. A partir desses métodos de conferência, existem três possibilidades para a identificação desse fuso, as quais estão resumidas na figura 12.6.

Figura 12.6: Métodos para se identificar o fuso horário empregado em dados do tipo `POSIXct` e `POSIXlt`

Referência do fuso horário

Quando um valor do tipo date-time aparece em seu console, uma referência do fuso horário associado a esse valor, aparece logo à direita do horário. Em muitas ocasiões, essa referência aparece como o código que identifica o fuso horário utilizado.

```
"2020-01-01 10:34:12 UTC"
```

Em outras ocasiões, ao invés de um código, essa referência do fuso horário é representada por um desvio numérico em relação ao fuso horário UTC.

```
"2020-01-01 10:34:12 +05"
```

Acesse o atributo tzzone

O atributo `tzzone` é responsável por armazenar uma referência do fuso horário associado a um determinado valor do tipo date-time. Caso este atributo esteja vazio, então o fuso horário associado ao objeto em questão, é o fuso horário padrão de seu sistema operacional, dado pela função `Sys.timezone()`.

```
vec <- as_datetime(
  "2020-01-01 10:34:12",
  tz = "Singapore"
)

attr(vec, "tzzone")

[1] "Singapore"
```

Fonte: Elaboração própria do autor.

Portanto, busque primeiro, reconhecer se alguma informação aparece à direita do horário. Se não há alguma informação nesse local, verifique o atributo `tzzone` desse objeto. Quando utilizamos as funções dos pacotes básicos do R para criar o nosso objeto, e não definimos algum fuso horário específico no argumento `tz`, o atributo `tzzone` do objeto resultante estará quase sempre vazio. Em casos como esse, o R vai automaticamente utilizar o fuso horário de seu sistema operacional, que pode ser identificado pelo resultado da função `Sys.timezone()`. Perceba abaixo, que o sistema operacional do meu computador, utiliza o horário de São Paulo, que é equivalente ao fuso horário de Brasília.

```
Sys.timezone()
```

```
## [1] "America/Sao_Paulo"
```

Esse fuso horário (de Brasília) se encontra a 3 desvios negativos do fuso horário UTC (vamos explicar mais a frente o que isso significa). Por esse motivo que, no exemplo abaixo, um -03 aparece ao final do valor de `vec`. Em outras palavras, essa informação (-03) está nos dizendo que o fuso horário empregado sobre o valor de `vec`, é o fuso horário que se encontra a 3 desvios negativos do fuso horário UTC, que pelos motivos apresentados acima, é o fuso horário de Brasília, ou, o fuso horário padrão do sistema operacional de meu computador.

```
vec <- as.POSIXct("2020-01-01 10:34:12")
attr(vec, "tzone")
```

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

```
vec
```

```
## [1] "2020-01-01 10:34:12 -03"
```

Em outras situações, o fuso horário presente à direita do horário será informado em seu código padrão. Veja o exemplo abaixo, em que utilizamos o fuso horário de Paris (França). Como resultado, o atributo `tzone` é preenchido pelo valor "Europe/Paris". Mas quando o valor de `vec` é desenhado em nosso console, o código CET é posicionado à direita do horário. Esse código se refere à *Central European Time* (ou “Tempo da Europa Central”), que é o fuso horário usufruído por diversos países europeus, incluindo a França.

```
vec <- as_datetime("2020-01-01 10:34:12", tz = "Europe/Paris")
attr(vec, "tzone")
```

```
## [1] "Europe/Paris"
```

```
vec
```

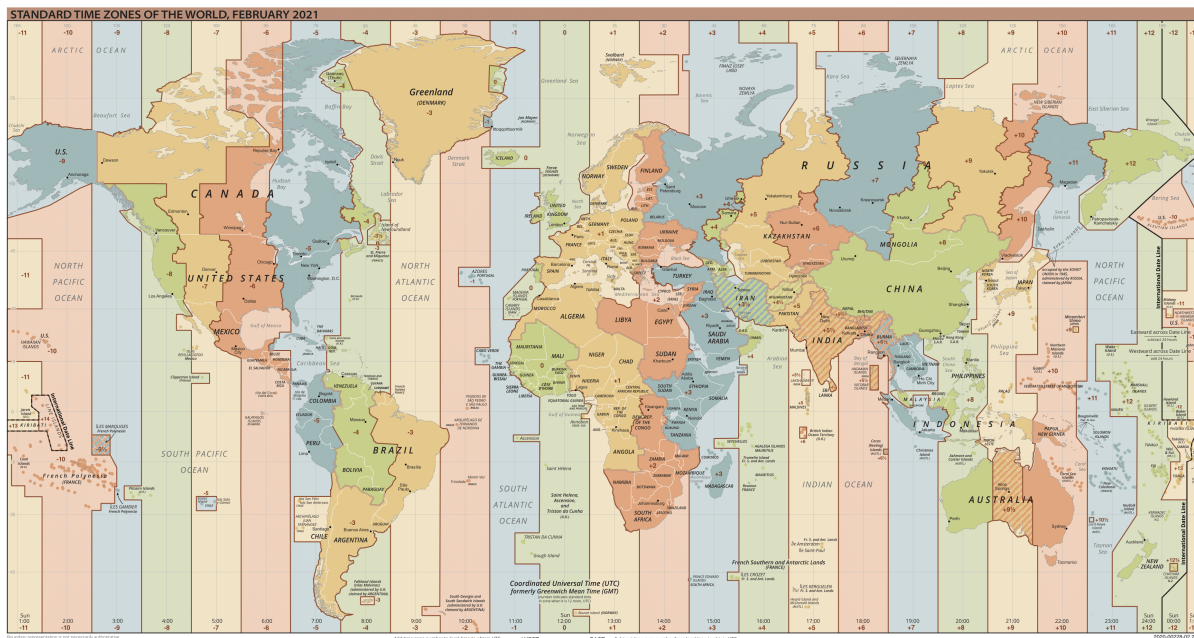
```
## [1] "2020-01-01 10:34:12 CET"
```

Sendo assim, você pode aplicar dois métodos diferentes sobre um valor do tipo `POSIXct` ou `POSIXlt`, para se identificar o fuso horário de referência. Para mais, compreenda que ao longo desses métodos, o fuso horário aplicado pode se apresentar em 3 formatos diferentes: por meio de um código (como CET, para *Central European Time*, ou UTC, para *Coordinated Universal Time*); por meio de uma região, ou uma cidade específica (como *America/Sao_Paulo*, ou *Europe/Paris*); ou então, por um desvio positivo ou negativo em relação ao fuso horário UTC (como -03, +05, ou +11).

12.7.2 Zonas horárias e o *Coordinated Universal Time* (UTC) como horário internacional

O planeta Terra é dividido em 24 zonas horárias, que são apresentadas na figura abaixo. No centro, se encontra a zona horária de número 0, que é a zona em que se encontra o famoso Meridiano de Greenwich, que por convenção, é o meridiano que divide a terra ao meio, ou, em outras palavras, que separa formalmente o oriente do ocidente. Cada zona horária, representa um fuso horário diferente, e, por isso, podemos dizer que há 24 fusos horários diferentes ao redor do mundo.

Figura 12.7: As diferentes zonas horárias presentes no planeta Terra



Fonte: Wikipedia.

Mesmo que cada zona horária seja determinada geograficamente, cada país ou cada nação tem o direito político de decidir qual a zona horária a qual ele pertence. Por esse motivo, mesmo que países como Argélia, Espanha e França estejam geograficamente sobre a zona horária de número 0, por decisão política própria, tais países foram incluídos na zona horária de número +1.

A zona horária de número 0, é a zona horária em que é calculado e utilizado o horário internacional, que é comumente denominado de *Coordinated Universal Time* (UTC), ou, Tempo Universal Coordenado. Sendo que **todos os fusos horários utilizados ao redor do mundo, são calculados a partir do horário UTC**. Dito de outra forma, UTC é o nome do fuso horário utilizado na zona horária de número 0, (isto é, a zona em que se encontra o Meridiano de Greenwich) e tal horário, é a base para determinarmos todos os outros horários empregados no mundo.

Sendo assim, o UTC é oficialmente o horário universal ou internacional utilizado no mundo. No passado, o sistema UTC era formalmente chamado de Tempo Médio de Greenwich, ou, *Greenwich Mean Time* (GMT), o qual sofreu alterações metodológicas importantes, que o transformaram no sistema UTC que conhecemos e utilizamos hoje. Por isso, caso você encontre algum horário acompanhado da sigla GMT, saiba que ele está se referindo ao “antigo UTC”.

12.7.3 Fusos horários como desvios do horário UTC

É muito importante destacar que, **o UTC é o horário internacional, não no sentido de que ele é o horário utilizado fora dos limites de qualquer país, mas sim, no sentido de que todos os outros fusos horários utilizados no planeta, são calculados a partir dele.**

Em resumo, o fuso horário aplicado em uma determinada zona horária, apresenta 1 hora de diferença em relação aos fusos horários empregados em suas zonas vizinhas. Tal efeito é de certa forma, uma consequência do fato da Terra levar aproximadamente 24 horas para dar a volta completa em torno de seu próprio eixo (esse movimento é chamado de rotação da Terra). Por esse motivo, cada uma das 24 zonas horárias possuem 15 graus de longitude (ou de “largura”). Pois a cada 15 graus que a Terra rotaciona, 1 hora se passa em nosso horário.

Essa afirmação pode ser posta matematicamente. Ao partirmos do princípio de que a Terra é uma esfera perfeita, sabemos que o planeta possui 360 graus de circunferência. Levando-se em conta que o planeta demora 24 horas para rotacionar-se em torno de seu próprio eixo, temos que $360/24 = 15$ graus por hora. Por essa razão que, cada zona horária, ou, cada fuso horário apresenta 1 hora de diferença em relação aos seus vizinhos.

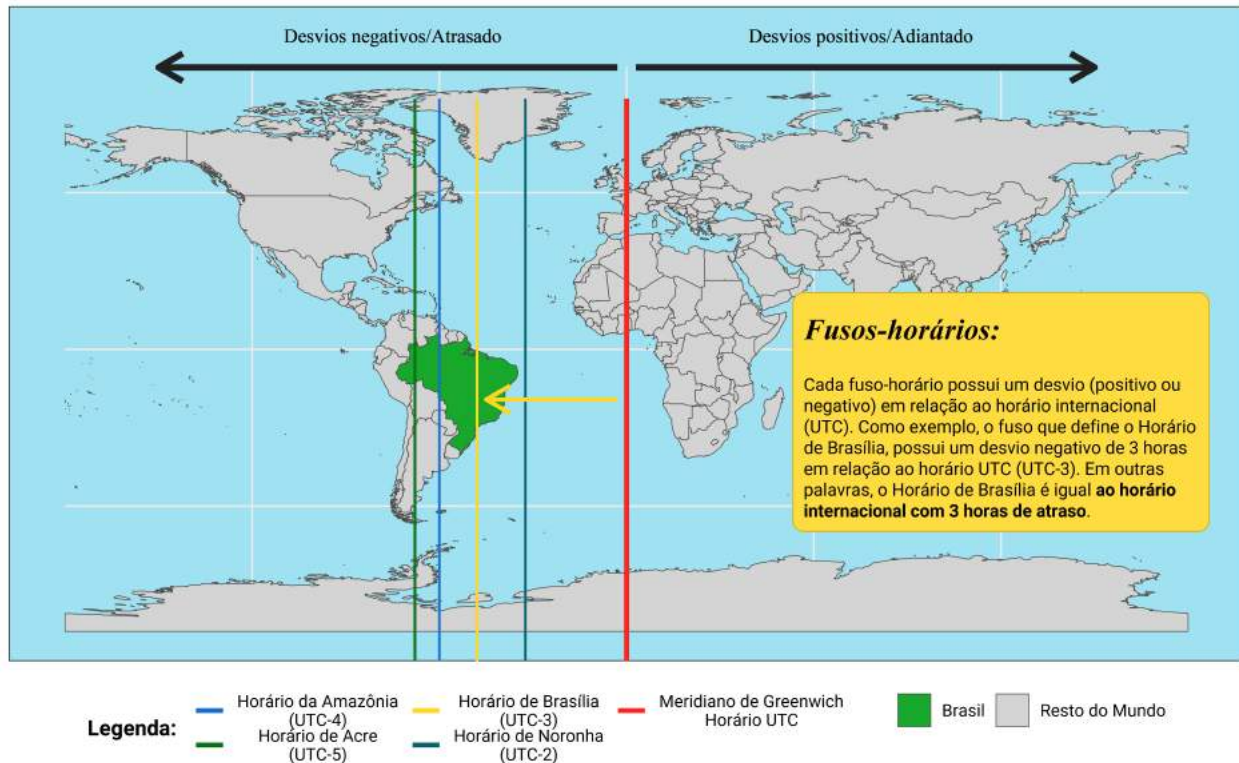
Com isso, podemos interpretar fusos horários como desvios de x horas em relação ao horário UTC (ou a zona horária de número 0). Ao analisarmos um determinado fuso horário, é muito importante identificarmos o lado do Meridiano de Greenwich (à esquerda ou à direita) no qual esse fuso se encontra, pois tal informação determinará se o desvio de x horas (em relação ao horário UTC) é negativo (à esquerda) ou positivo (à direita). Se o desvio for negativo, significa que o desvio deve ser subtraído do horário internacional (ou seja, o horário do país em questão, está atrasado em relação ao horário UTC). Por outro lado, se o desvio for positivo, significa que esse desvio deve ser acrescido ao horário internacional (o país está com horário adiantado).

Portanto, um fuso horário é calculado a partir de um desvio de x horas em relação ao horário UTC. Para sabermos o número x de horas a serem descontadas (ou adicionadas) do horário UTC, temos que saber a distância da zona horária em análise da zona horária de número 0. Por exemplo, ao voltarmos para a figura 12.7, podemos identificar que a Finlândia está inclusa na zona horária de número +2 e, por estar a duas zonas horárias de distância da zona horária de número 0, sabemos que o horário empregado na Finlândia possui um desvio positivo de 2 horas em relação ao horário UTC. Isso significa que a Finlândia está 2 horas adiantada em relação ao horário internacional.

Como um outro exemplo, o Brasil participa de 4 zonas horárias diferentes (de números -2, -3, -4 e -5). Logo, o Brasil possui 4 fusos horários diferentes ao longo de suas regiões, sendo o fuso horário de Brasília o mais comum dentre eles. A zona horária que representa o fuso horário de Brasília, é

a zona de número -3. Isso significa que o horário de Brasília está a 3 desvios negativos do horário UTC, ou, dito de outra forma, o horário de Brasília é equivalente ao horário internacional subtraído de 3 horas (ou atrasado em 3 horas). Tal relação está exposta pela figura 12.8.

Figura 12.8: O Brasil possui quatro fusos horários diferentes



Fonte: Elaboração própria do autor.

12.7.4 Quando fusos horários diferentes geram uma grande confusão

Uma das principais características do pacote lubridate é a de que suas funções tentam utilizar o fuso horário UTC em todo lugar. Por isso, em todas as ocasiões em que não definirmos explicitamente um fuso horário a ser empregado no argumento `tz` de `as_datetime()`, ou de `dmy_hms()`, o valor resultante dessas funções vai utilizar o fuso UTC.

No entanto, as funções dos pacotes básicos do R adotam um protocolo diferente. Ao não definirmos um fuso horário no argumento `tz` das funções `as.POSIXct()` e `strptime()`, o fuso horário padrão do sistema operacional será automaticamente empregado sobre o resultado. No caso do Brasil, enfrentamos 4 fusos horários diferentes. Logo, a depender de onde você se encontra no país, você

talvez tenha resultados diferentes dos que são apresentados a seguir. Mas o raciocínio permanece o mesmo, independente de onde você se encontra no planeta.

Na prática, essa diferença entre padrões só impacta em seus resultados, caso você esteja trabalhando com diversos fusos horários ao mesmo tempo, ou, se você deseja aplicar alguma operação que lida diretamente com o fuso horário de um dado do tipo `date-time`. Um exemplo de operação que lida diretamente com o fuso de referência dos dados e, que, portanto, possui diferentes resultados entre `as_datetime()` e `as.POSIXct()`, é o uso do código `%z` em `format`, para incluir um desvio (em horas e minutos) sobre o fuso horário a ser aplicado sobre o resultado final.

Na seção **O que devo fazer se meus dados se encontram em um formato diferente?**, demos um exemplo prático que demonstra esse efeito, que nasce da diferença entre os fusos horários padrões adotados pelas funções. Nesse exemplo possuíamos um vetor chamado `dt`, contendo o instante: 8 horas do dia 01 de janeiro de 2020. O objetivo principal desse exemplo era demonstrar que, se não definirmos algum fuso horário no argumento `tz`, as funções `as_datetime()` e `as.POSIXct()` nos trazem resultados diferentes, quando aplicadas sobre os mesmos valores do tipo `date-time`. Tal exemplo está reproduzido abaixo:

```
dt <- "2020-01-01 08:00:00 -0300"
as.POSIXct(dt, format = "%F %T %z")

## [1] "2020-01-01 08:00:00 -03"

as_datetime(dt, format = "%F %T %z")

## [1] "2020-01-01 11:00:00 UTC"
```

Lembre-se que o valor `-0300` presente ao final do *string* armazenado em `dt`, representa um desvio negativo de 3 horas que será interpretado pelo código `%z`. Para mais, lembre-se que o valor `-03` presente ao final do resultado de `as.POSIXct()` representa apenas o fuso horário empregado nesse resultado e, portanto, não possui qualquer relação com o desvio de `-0300` do código `%z`.

Primeiro, ao observarmos o resultado de `as.POSIXct()`, percebemos que o desvio de 3 horas (`-0300`) não gerou alterações no horário (8 horas em ponto) contido em `dt`. Entretanto, nós também podemos observar abaixo, que a função `as_datetime()` “adicionou” esse desvio ao valor presente em `dt`, gerando assim, um horário adiantado em 3 horas. A lógica por trás desses resultados, começa a ficar mais clara, a medida em que alteramos o valor do desvio, como demonstrado abaixo.

```
dt <- "2020-01-01 08:00:00 -0200"
as.POSIXct(dt, format = "%F %T %z")

## [1] "2020-01-01 07:00:00 -03"

as_datetime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 10:00:00 UTC"
```

Como um outro exemplo, podemos alterar o sinal do desvio. Porém, ao contrário do que você provavelmente está pensando, mesmo um desvio positivo acaba sendo subtraído do horário. Dessa vez, `as.POSIXct()` subtraiu 5 horas do horário original de `dt`, enquanto em `as_datetime()`, a redução foi de apenas 2 horas.

```
dt <- "2020-01-01 08:00:00 +0200"
as.POSIXct(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 03:00:00 -03"
```

```
as_datetime(dt, format = "%F %T %z")
```

```
## [1] "2020-01-01 06:00:00 UTC"
```

A medida em que testamos diferentes valores para esse desvio, podemos perceber que a adição do desvio representado pelo código `%z` segue a fórmula:

$$H = h - [(d \times -1) + z]$$

Sendo que, as variáveis presentes nessa equação são:

- H : hora presente no resultado final da função.
- h : hora inicial, ou, em outras palavras, a hora que está presente no *string* de *input*.
- d : número de desvios (em relação ao fuso UTC) que representa o fuso horário empregado no resultado final da função.
- z : o valor do desvio presente no *string* de *input*, e que é representado pelo código `%z`.

Desse modo, ao olharmos para o *string* armazenado em `dt`, identificamos que o valor de h nessa equação, corresponde a 8 horas. Como nós não alteramos o horário presente nesse *string* em nenhum dos exemplos anteriores, o valor de h esteve sempre fixo. O que estava variando de um exemplo para o outro, era o valor de z e o valor de d .

A valor da variável d depende apenas de qual o fuso horário adotado pela função que estamos utilizando. Quando utilizamos a função `as_datetime()`, o valor de d será igual a zero, pois essa função sempre tenta adotar o fuso horário UTC em seus resultados. Contudo, quando utilizamos as funções dos pacotes básicos do R, o valor de d vai depender de qual é o fuso horário padrão de seu sistema operacional. No meu caso, o valor de d em meu computador (quando a função `as.POSIXct()` ou `strptime()` é empregada) é igual a -3 (que é o desvio que representa o fuso horário de Brasília).

Tendo essas informações em mente, podemos concluir que a diferença entre os resultados das funções `as.POSIXct()` e `as_datetime()` se deve apenas à divergência entre os fusos horários adotados por cada função, o que impacta diretamente no valor da variável d para cada função. Em outras palavras, se H_{lubr} e H_{base} são as horas calculadas por `as_datetime()` e `as.POSIXct()`, e, se d_{lubr} e

d_{base} são os desvios que representam os fusos horários adotados por cada função, respectivamente, podemos expor essa diferença de forma matemática:

$$\begin{aligned}H_{lubr} &= 8 - [(d_{lubr} \times -1) + z] = 8 + d_{lubr} - z \\H_{base} &= 8 - [(d_{base} \times -1) + z] = 8 + d_{base} - z \\H_{base} - H_{lubr} &= 8 + d_{base} - z - (8 + d_{lubr} - z) \\H_{base} - H_{lubr} &= d_{base} - d_{lubr}\end{aligned}$$

Com essas equações, podemos rapidamente identificar que se ambas as funções aplicarem o mesmo fuso horário, as variáveis d_{lubr} e d_{base} serão iguais e, conseqüentemente, essa diferença entre H_{base} e H_{lubr} desaparece. Como exemplo, perceba abaixo que ambas as funções retornam o mesmo resultado, ao escolhermos um fuso horário específico no argumento `tz` de cada função, como por exemplo, o horário de Toronto (Canadá).

```
dt <- "2020-01-01 08:00:00 +0200"
### Quando ambas as funções utilizam o mesmo fuso horário
### a inserção do código %z gera o mesmo resultado
as.POSIXct(dt, format = "%F %T %z", tz = "America/Toronto")

## [1] "2020-01-01 01:00:00 EST"

as_datetime(dt, format = "%F %T %z", tz = "America/Toronto")

## [1] "2020-01-01 01:00:00 EST"
```

Portanto, fique atento a possíveis diferenças entre os horários que resultam de funções que, em tese, deveriam ser “equivalentes” e, que portanto, deveriam gerar os mesmos resultados. Essa seção, buscou demonstrar que tal diferença pode nascer da divergência entre os fusos horários adotados por cada função. Caso você encontre uma diferença dessa natureza, busque pela documentação interna de cada função, e procure entender como essas funções trabalham com o fuso horário.

12.7.5 Interpretando um mesmo ponto no tempo em diferentes fusos horários

Vamos construir mentalmente duas pessoas. Cláudio mora e trabalha no Rio de Janeiro, e tem feito alguns projetos internacionais na área de marketing. Um de seus principais parceiros é Ryuichi, um grande empresário do Japão. Suponha que Cláudio e Ryuichi tenham marcado uma reunião entre eles, às 9hrs da manhã no horário do Japão (isto é, no horário local para o Ryuichi). Qual será o horário da reunião no Brasil? Ou seja, que horas Cláudio deve ligar o seu computador e acessar a sala de reunião para conversar com Ryuichi?

Podemos rapidamente responder a essa questão, com a função `with_tz()`. Precisamos primeiro, criar um objeto que guarde o horário de 9hrs segundo o fuso horário do Japão e, em seguida, pedimos

à função, que nos mostre esse mesmo instante segundo o horário de São Paulo. Como você pode ver abaixo, Cláudio teria que entrar na reunião às 21hrs do dia anterior ao dia marcado por Ryuichi.

```
horario_japao <- ymd_hm("2020-01-01 09:00", tz = "Japan")
with_tz(horario_japao, "America/Sao_Paulo")
```

```
## [1] "2019-12-31 21:00:00 -03"
```

Portanto, o horário do Japão está 12 horas adiantado em relação ao horário utilizado por Cláudio. Isso significa que, poderíamos chegar ao mesmo resultado mostrado pela função `with_tz()`, ao subtraírmos 12 horas do valor presente `horario_japao`. Lembre-se que, valores do tipo `POSIXct` são armazenados em segundos, logo, para subtraírmos as 12 horas, precisamos multiplicar essas 12 horas com os 60 minutos (de cada hora) e com os 60 segundos (de cada minuto).

```
horario_japao - (12 * 60 * 60)
```

```
## [1] "2019-12-31 21:00:00 JST"
```

No entanto, ao invés de adicionar ou subtrair segundos, no fundo, o que a função `with_tz()` faz, é retornar o mesmo objeto contendo um atributo `tzone` diferente. Em outras palavras, podemos ainda chegar ao mesmo resultado de `with_tz()`, ao alterarmos o valor do atributo `tzone` em `horario_japao` para o fuso horário de Cláudio, como demonstrado abaixo.

```
horario_brasil <- horario_japao
attr(horario_brasil, "tzone") <- "America/Sao_Paulo"
horario_brasil
```

```
## [1] "2019-12-31 21:00:00 -03"
```

Por isso, fique atento aos seus dados do tipo `POSIXct` e `POSIXlt`. Na hipótese desses dados se alterarem repentinamente, sem alguma explicação clara, pode ser que alguma operação que você realizou tenha alterado o valor conectado ao atributo `tzone` desses dados e, com isso, provocado esse efeito.

12.8 Calculando intervalos com o tipo `difftime`

O R oferece de forma nativa, um outro tipo de variável de tempo que é útil para calcularmos intervalos ou diferenças entre dois pontos no tempo. Esse tipo é comumente chamado de `difftime`, e é representado principalmente pela função `difftime()`.

O tipo `difftime` é na verdade, um tipo de dado muito simples. Em resumo, um dado do tipo `difftime` é um dado do tipo `double`, acompanhado de um atributo chamado `units`, que guarda a

unidade na qual o valor `double` se encontra. Porém, o papel que esse tipo busca cumprir não é nada simples.

Por exemplo, vamos supor dois horários em um mesmo dia, como 09 horas e 16 horas. A diferença entre esses dois pontos é de 7 horas. Tudo que o tipo `difftime` faz é, armazenar a unidade “horas” no atributo `units` que está conectado ao número 7. Dito de outra forma, um dos papéis que o tipo `difftime` cumpre é manter o controle das unidades de tempo empregadas em valores que representam um intervalo de tempo (ou a duração de algum evento).

```
dt1 <- ymd_h("2020-01-01 09")
dt2 <- ymd_h("2020-01-01 16")
```

```
difftime(dt2, dt1)
```

```
## Time difference of 7 hours
```

Devido a esse controle, o tipo `difftime` é capaz de eficientemente calcular o intervalo de tempo, entre valores que se encontram em unidades de tempo diferentes. Por exemplo, qual a diferença entre 14 horas e 14000 segundos? Ao convertermos esses números para valores do tipo `difftime`, o R se torna capaz de identificar as unidades de cada um. Dessa forma, o R pode reconhecer qual a maneira ideal de converter ambos os valores para a mesma unidade, e com isso, calcular corretamente a diferença entre os dois.

```
horas <- as.difftime(14, units = "hours")
segundos <- as.difftime(14000, units = "secs")
```

```
horas - segundos
```

```
## Time difference of 36400 secs
```

Além disso, a função `difftime()` lhe permite escolher a unidade que você deseja para o resultado. Logo, se você deseja saber quantas semanas estão entre as datas 14 de março de 2020 e 01 de janeiro de 2020, você pode rapidamente calcular esse valor da seguinte maneira:

```
janeiro <- ymd("2020-01-01")
marco <- ymd("2020-03-14")
```

```
difftime(marco, janeiro, units = "weeks")
```

```
## Time difference of 10.42857 weeks
```

Caso você não precise de um nível de precisão muito elevado, você aplicar funções como `round()`, `ceiling()` e `floor()` sobre o resultado de `difftime()`. Por exemplo, a parte decimal do valor que calculamos acima (10,42857 semanas) é de pouco valor para nós. Digo, quanto é 0,42857 ou

42,857% de uma semana? Por essa questão, seria interessante aplicarmos a função `floor()` sobre este resultado, para sabermos qual o número de semanas completas que existem entre as datas de marco e janeiro.

```
floor(difftime(marco, janeiro, units = "weeks"))
```

```
## Time difference of 10 weeks
```

12.9 Como as variáveis de tempo são interpretadas pelo R ?

Em resumo, qualquer informação que represente uma data (ex: 20/12/2020) é geralmente interpretada pelo R por meio do tipo `Date`; já datas que são acompanhadas de algum horário (ex: 20/12/2020 10:32:41) são assimiladas pelo R através dos tipos `POSIXlt` e `POSIXct` (PENG, 2015); e, por último, quando temos a duração de algum evento, ou principalmente, a diferença de tempo entre duas datas (ex: a diferença entre 12 de Março e 15 de Março é de três dias, ou, 72 horas, ou, 4.320 minutos, ou, 259.200 segundos), temos a opção de empregarmos o tipo `difftime` sobre essas informações (mas nem sempre isso é necessário).

No fundo, qualquer dado que for interpretado pelos tipos `Date`, `POSIXlt`, `POSIXct`, ou `difftime`, é armazenado pelo R como um número real, isto é, um dado do tipo `double`. Ou seja, da mesma forma em que descrevemos (no capítulo anterior) o tipo `factor` como um “parente” do tipo básico `integer`, os tipos `Date`, `POSIXlt`, `POSIXct` e `difftime` são na realidade, parentes do tipo básico `double`, ou, dito de outra forma, são construídos a partir dele. E o que diferencia esses tipos do tipo básico `double`, são as suas classes e atributos.

Assim sendo, em termos técnicos, podemos dizer que um dado que se encontra no tipo `Date`, `POSIXlt`, `POSIXct`, ou `difftime`, é na verdade, um dado do tipo `double` que possui classe `Date`, `POSIXlt`, `POSIXct`, ou `difftime`, respectivamente. Para mais, um objeto que se encontra no tipo `POSIXlt` ou `POSIXct`, inclui um atributo chamado `tzzone`. Já um objeto do tipo `difftime`, possui um atributo chamado `units`. Dito de outra forma, os tipos `Date`, `POSIXlt`, `POSIXct` e `difftime` são armazenados por meio do tipo `double`, mas apresentam diferentes classes e atributos que os diferenciam uns dos outros.

Isso significa que, por exemplo, para testarmos corretamente se um objeto do R se encontra no tipo `Date`, nós devemos aplicar um teste lógico parecido com o teste abaixo. Ambos os vetores (`double_vec` e `date_vec`) conseguem passar (isto é, adquirem um valor `TRUE`) na primeira parte do teste (`is.double(x)`), pois ambos os vetores são do tipo `double`. Entretanto, apenas o vetor `date_vec` é capaz de passar também na segunda parte do teste (`class(x) == "Date"`), pois apenas `date_vec` possui classe `Date`.

```
double_vec <- c(0.5, 1.2, 1.5, 2.4)
date_vec <- as.Date(c("2020-09-10", "2020-09-11", "2020-09-12"))

is.double(double_vec) & class(double_vec) == "Date"
```

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

```
is.double(date_vec) & class(date_vec) == "Date"
```

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

Uma segunda forma mais direta de realizarmos esse teste é através da função `inherits()`, que é capaz de identificar se um objeto específico do R “herda”, ou apresenta as características específicas de um tipo em questão.

```
inherits(double_vec, "Date")
```

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

```
inherits(date_vec, "Date")
```

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

12.9.1 Escala de tempo e o conceito de data de origem

Para que essa característica fique clara, veja o exemplo abaixo. Primeiro, eu guardo a data 10 de março de 2020 no objeto `d`. Ao questionarmos o R sobre o tipo de dado e classe utilizados pelo objeto `d`, vemos que ele é um vetor do tipo `double` com classe `Date`. Em seguida, eu aplico a função `unclass()` sobre o objeto `d`, para que o R me mostre exatamente como essa data está sendo armazenada.

Repare abaixo, que o R está guardando a data 2020-03-10 (quer dizer, 10 de março de 2020) como o número 18331. Agora, você provavelmente está se questionando: o que esse número 18331 significa? Como ele é traduzido para a data 10 de março de 2020? Essas questões são respondidas pelo conceito de **data de origem**.

```
## O objeto d guarda a data
```

```
## 10 de março de 2020
```

```
d <- as.Date("2020-03-10")
```

```
typeof(d)
```

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

```
class(d)
```

```
## [1] "Date"
```

```
unclass(d)
```

```
## [1] 18331
```

Este número, pelo qual o R guarda a data 2020-03-10, representa o número de dias decorridos desde a data de origem até a data 2020-03-10. Digo, 18331 dias se passaram desde a data de origem até atingirmos o dia 10 de março de 2020. Portanto, a **data de origem representa o “marco zero”, ou o ponto zero da escala de tempo**, e podemos descobrir qual é essa “data de origem” utilizada pelo R, ao subtrairmos da data armazenada o número que a representa.

Por exemplo, se o número 18331 representa a data 2020-03-10, ao subtrairmos esse número dessa data, o R acaba nos retornando a data 1970-01-01. Portanto, o “dia zero” segundo a perspectiva do R, é o dia 01 de janeiro de 1970. Isso significa que, todos os seus dados no R que estiverem sendo interpretadas pelo tipo Date, vão ser (obrigatoriamente) armazenados pelo R como o número de dias entre a data de origem do R (o dia 01 de janeiro de 1970) e as suas datas em questão.

```
as.Date("2020-03-10") - 18331
```

```
## [1] "1970-01-01"
```

Essa característica é muito importante, e não é particular ao R. Diversas linguagens de programação, e programas comuns (como o Excel) implementam variáveis de tempo desta mesma maneira⁴. Logo, no R, qualquer informação que descreva um ponto específico do tempo é armazenada como um número, e, tal número representa (de certa maneira) uma “medida de distância” entre o ponto zero da escala de tempo e o ponto do tempo com o qual você está trabalhando.

Tendo isso em mente, qual é o número que representa a data 10 de janeiro de 1970 no R? Se você compreendeu os conceitos apresentados nessa seção, você certamente respondeu que esse valor é o número 9. Pois partindo do dia 01 de janeiro de 1970 até o dia 10 de janeiro, temos 9 dias de diferença.

```
d <- as.Date("1970-01-10")
unclass(d)
```

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

Mas e as datas anteriores ao dia 01 de janeiro de 1970? Como o R representa essas datas? Mesmo nessa situação, o R não muda o seu comportamento. Contudo, como essas datas se encontram atrás do “ponto zero” na escala do tempo utilizada, o R vai representar essas datas com números negativos. Por exemplo, o dia 30 de dezembro de 1969 é representado por meio do número -2. Pois essa data se encontra a 2 dias atrás do dia 01 de janeiro de 1970.

⁴Portanto, diversos programas e linguagens utilizam números para representar pontos em uma “escala de tempo”. Porém, o que tende a divergir e muito entre esses diversos sistemas é a data de origem utilizada (GROTHENDIECK; PETZOLDT, 2004). O Excel por exemplo, utiliza o dia 01 de janeiro de 1900 como o ponto zero de sua escala de tempo, enquanto o SPSS, utiliza o dia de início do calendário gregoriano (14 de outubro de 1582).

```
d <- as.Date("1969-12-30")
unclass(d)
```

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

Portanto, quando você aplica uma ordenação sobre um vetor do tipo `Date`, `POSIXlt`, `POSIXct` ou `difftime`, ao invés de o R comparar o dia, mês e ano de cada data, ele vai utilizar os números que representam cada ponto no tempo presente nesse vetor para calcular essa ordenação. Em outras palavras, esses valores são ordenados de acordo com as suas distâncias em relação à data de origem utilizada pelo R.

12.9.2 A unidade ou a escala de tempo muda com o tipo de dado

Nós sabemos que o tipo `Date` é armazenado como o número de dias em relação à data de origem. Porém, um choque ocorre quando tentamos transportar isso para os tipos `POSIXlt` e `POSIXct`. Pois qualquer dado que for interpretado por algum desses dois tipos, vai ser armazenado como os segundos em relação ao ponto de origem.

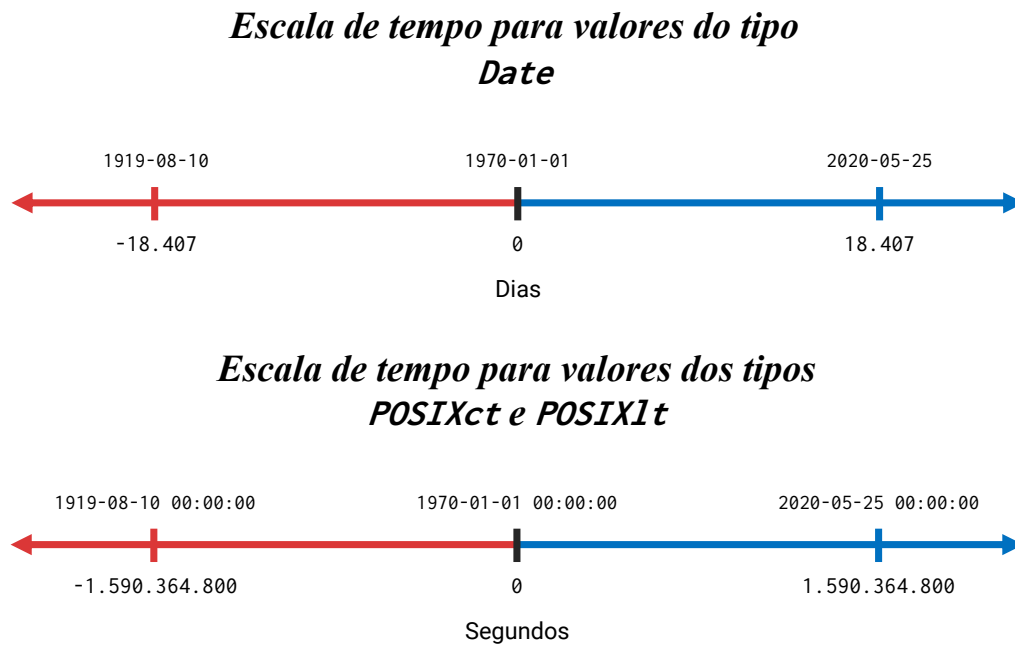
Ou seja, a teoria continua a mesma; a sua informação continua sendo armazenada como um número, que representa uma “medida de distância” até o “ponto zero” da escala de tempo. Entretanto, a unidade utilizada nessa escala de tempo muda de acordo com o tipo de dado que você emprega. Logo, quando estamos discutindo o tipo `Date`, o R recorre à uma escala de tempo **em dias**. Mas quando estamos nos referindo aos tipos `POSIXlt` e `POSIXct`, essa mesma escala de tempo é interpretada **em segundos**.

Para mais, a data de origem é a mesma em ambas as escalas (01 de janeiro de 1970). Todavia, como os tipos `POSIXlt` e `POSIXct` são capazes de guardar horários, esses tipos vão utilizar um ponto específico dessa data de origem como referência. Isto é, ao invés de utilizar o dia 01 de janeiro de 1970 como um todo, os tipos `POSIXlt` e `POSIXct` empregam **um ponto, ou, um horário específico** desse dia como o ponto zero de sua escala de tempo. E esse horário é, de certa forma, o ponto zero desse dia, ou, de outra forma, a meia noite desse dia no fuso horário UTC. Logo, o **ponto de origem** na escala da qual os tipos `POSIXlt` e `POSIXct` usufruem é o horário 00:00:00 do dia 01 de janeiro de 1970, especificamente no fuso horário UTC.

Com isso, se o dia 2020-03-10 está a 18331 dias de distância do dia 1970-01-01, a quantos segundos o horário 14:30 desse mesmo dia (2020-03-10 14:30:00) está de distância do ponto zero da escala (1970-01-01 00:00:00)? Para descobriremos a resposta, podemos aplicar o mesmo método que utilizamos com o tipo `Date`, por meio da função `unclass()`. Vemos abaixo, que tal horário é interpretado pelo R como o segundo 1583850600. Em outras palavras, esse horário está aproximadamente a 1,583 bilhão de segundos de distância do ponto zero da escala.

```
hr <- as.POSIXct("2020-03-10 14:30:00", tz = "UTC")
unclass(hr)
```

Figura 12.9: Representação visual da escala de tempo utilizada por cada tipo de dado



Fonte: Elaboração própria do autor.

```
## [1] 1583850600
## attr(,"tzone")
## [1] "UTC"
```

Exercícios

Exercício 1

Cada item abaixo pode lhe pedir para identificar a origem de algum erro, ou de algum resultado inesperado, ou ainda, requisitar que você trabalhe com algum objeto específico para um dado fim.

- A) Perceba abaixo, que ao transformarmos o vetor `vec` para o tipo `Date`, alguns elementos são transformados para valores `NA`. Porque essa transformação ocorre? Há alguma solução clara para isso?

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

```
as.Date(vec)
```

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

- B) Que comandos você utilizaria para transportar o vetor `vec` abaixo para o tipo `Date`?

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

- C) Como definimos neste capítulo, no R, dados do tipo `date-time` são armazenados como o número de segundos desde a meia noite de 01 de janeiro de 1970. Porém, por alguma razão inesperada, quando eu crio um objeto contendo este exato ponto no tempo, e retiro a sua classe com a função `unclass()`, percebo que este ponto foi armazenado como o valor 10800. Porque isso ocorre? Ele não deveria ser armazenado como zero?

```
ponto <- as.POSIXct("1970-01-01 00:00:00")
unclass(ponto)
```

```
## [1] 10800
## attr(,"tzone")
## [1] ""
```

Exercício 2

Como definimos anteriormente neste capítulo, diversos programas, incluindo o Excel, armazenam valores do tipo `date-time` como o número de dias ou de segundos, em relação a um ponto específico de origem na escala do tempo. Logo abaixo, temos a tabela `dados_excel`. Essa tabela nos apresenta na coluna `como_numero`, o número aproximado no Excel que representa os valores do tipo `date-time` presentes na coluna `como_data`. Ou seja, no Excel, o ponto "20/02/2020 03:45:00" é armazenada como o número decimal 43.881,15625. Considerando que, no sistema Windows, o Excel utiliza a data 30 de Dezembro de 1899 (ou "1899-12-30") como o seu ponto de origem, o seu trabalho nessa questão é converter os números presentes no vetor `numero_no_excel` para o

tipo POSIXct, de modo que o resultado contenha os mesmos instantes apresentados no vetor `datetime_no_excel`. **Dica:** configure o argumento `tz` para o fuso horário UTC, dessa forma, você evita em sua conversão, possíveis adições/subtrações automáticas que emergem da diferença entre o fuso de seu sistema operacional e o fuso UTC.

```
datetime_no_excel <- c(
  "20/02/2020 03:40:00",
  "20/02/2020 03:45:00",
  "20/02/2020 03:50:00",
  "20/02/2020 03:55:00",
  "20/02/2020 04:00:00"
)

numero_no_excel <- c(
  43881.1527777778,
  43881.15625,
  43881.15972222226,
  43881.1632060185,
  43881.1666666667
)

dados_excel <- data.frame(
  como_data = datetime_no_excel,
  como_numero = numero_no_excel
)

print(dados_excel)
```

##	como_data	como_numero
## 1	20/02/2020 03:40:00	43881.15
## 2	20/02/2020 03:45:00	43881.16
## 3	20/02/2020 03:50:00	43881.16
## 4	20/02/2020 03:55:00	43881.16
## 5	20/02/2020 04:00:00	43881.17

Respostas dos exercícios de cada capítulo

Capítulo 1 - Noções Básicas do R

Exercício 1

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

```
32397 + 55405
```

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

```
### Ou de forma análoga:
```

```
n1 <- 32397
```

```
n2 <- 55405
```

```
n1 + n2
```

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

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

```
### Lembre-se de criar o objeto conj
```

```
conj <- c(290, 34, 512, 54, 89)
```

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

```
print(resposta)
```

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

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

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

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

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

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

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

Exercício 2

1.2.A) Qualquer erro do tipo “objeto ‘x’ não encontrado”, significa que o objeto pelo qual você requisitou não existe atualmente em sua seção do R. Portanto, o erro na questão que diz respeito a um objeto chamado *logica*, está lhe dizendo que você não criou ainda nenhum objeto chamado *logica* em sua sessão. A partir do momento em que você define um valor para o nome *logica*, esse erro passa a não ocorrer mais. Lembre-se que esse erro pode surgir **em qualquer lugar** (dentro de qualquer função, ou de qualquer operação), pois sem esse objeto (que não existe em seu computador) o R não é capaz de completar a operação pela qual você requisitou.

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

```
sum(logica)
```

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

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

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

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

Logo, o erro disposto na questão surge quando tentamos acessar a função `bind_rows()`, quando ela ainda não foi carregada para a nossa seção do R. Como comentamos ao longo da seção **Pacotes**, para

acessarmos as funções disponíveis dentro de um pacote, é necessário carregarmos esse pacote para a nossa seção (através da função `library()`). E para carregarmos um pacote para a nossa seção, é necessário que esse pacote esteja instalado em nosso computador.

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

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

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

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

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

Exercício 3

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

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

Sendo que, as variáveis nessa equação são:

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

- σ : desvio padrão da variável X .

Logo, os comandos necessários para o cálculo são:

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

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

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

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

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

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

Sendo que, as variáveis nessa equação são:

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

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

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

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

## [1] 0.9
```

Capítulo 2 - Fundamentos da Linguagem R

Exercício 1

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

```
list(1:30)

## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
```

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

```
1:10

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

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

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

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

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

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

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

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

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

2.1.F) Um vetor atômico preenchido por 25 NA's.

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

Exercício 2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

"estado" %in% names(lst)

## [1] TRUE

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

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

"estado" %in% names(lst_sem_estado)

## [1] FALSE
```

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

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



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

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

2.2.E) Lembre-se que as três condições apresentadas no enunciado da questão são dependentes.

Logo, o objeto que você está testando deve se encaixar nas três condições ao mesmo tempo. Em termos técnicos, isso significa que os testes lógicos referentes a cada uma das três condições, devem obrigatoriamente retornar TRUE para este objeto. Se ao aplicarmos pelo menos um desses testes, e o resultado for FALSE, isso significa que o objeto em questão não se encaixa no teste lógico com um todo, ou, o resultado geral do teste lógico é FALSE. Para que o R entenda que essas condições são dependentes, e que devem ser satisfeitas ao mesmo tempo, você deve conectar as três condições pelo operador &.

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

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

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

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

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

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

```
## [1] TRUE

"vendas" %in% colnames(tab)

## [1] TRUE

is.character(tab[[3]])

## [1] FALSE
```

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

Para identificarmos se um dado número é múltiplo de um outro número, podemos observar se o resto da divisão entre esses dois números é igual a zero. Em outras palavras, se desejamos saber que um dado valor x é múltiplo de um dado valor y , podemos realizar o cálculo $x \div y$ e, observar se o resto dessa divisão é ou não igual a zero. Lembre-se que no R, temos o operador aritmético `%`, que nos retorna justamente o resto da divisão entre dois números.

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

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

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

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

## [1] TRUE
```

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

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

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

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

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

Com isso, precisamos apenas conectar esse componente ao outro componente `(ano %% 4 == 0)`, que representa a condição 1, formando assim a estrutura final do teste: `(ano %% 4 == 0) & ((ano %% 100 != 0) | (ano %% 400 == 0))`. Dessa vez, utilizamos o operador que indica dependência (&) entre esses dois componentes principais do teste lógico, que são `(ano %% 4 == 0)` e `(ano %% 100 != 0) | (ano %% 400 == 0)`, pois um número que é múltiplo de 4, ainda pode ser um múltiplo de 100. Portanto, essa condição de dependência apenas assegura que a condição 2 seja respeitada, caso o número atenda a condição 1.

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

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

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

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

Exercício 3

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

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

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

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

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

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

## [1] "double"
```

2.3.C) O vetor resultante será do tipo double

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

## [1] "double"
```

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

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

## [1] "integer"
```

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

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

## [1] "character"
```

Exercício 4

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

```
library(nycflights13)

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

flights[teste, ]
```

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

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

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

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

flights[teste, ]
```

```
## # A tibble: 95,685 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <dbl>         <dbl>       <dbl>   <dbl>         <dbl>
## 1  2013     1     1     533             529          4     850             830
## 2  2013     1     1     542             540          2     923             850
## 3  2013     1     1     559             600         -1     941             910
## 4  2013     1     1     608             600          8     807             735
## 5  2013     1     1     611             600         11     945             931
## 6  2013     1     1     624             630         -6     909             840
## 7  2013     1     1     628             630         -2    1016             947
## 8  2013     1     1     632             608         24     740             728
## 9  2013     1     1     635             635          0    1028             940
## 10 2013     1     1     702             700          2    1058             1014
## # ... with 95,675 more rows, and 11 more variables: arr_delay <dbl>,
```

```
## # carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## # air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

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

```
library(nycflights13)
```

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

```
flights[teste, ]
```

```
## # A tibble: 11,500 x 19
```

```
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <dbl>         <dbl>      <dbl>    <dbl>         <dbl>
## 1  2013     1     1     554           600         -6      812           837
## 2  2013     1     1     555           600         -5      913           854
## 3  2013     1     1     557           600         -3      709           723
## 4  2013     1     1     557           600         -3      838           846
## 5  2013     1     1     558           600         -2      753           745
## 6  2013     1     1     558           600         -2      849           851
## 7  2013     1     1     558           600         -2      853           856
## 8  2013     1     1     558           600         -2      924           917
## 9  2013     1     1     558           600         -2      923           937
## 10 2013     1     1     559           600         -1      941           910
```

```
## # ... with 11,490 more rows, and 11 more variables: arr_delay <dbl>,
## # carrier <chr>, flight <dbl>, tailnum <chr>, origin <chr>, dest <chr>,
## # air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## # time_hour <dtm>
```

```
### Ou de forma um pouco mais organizada:
```

```
###
```

```
condicao1 <- flights$arr_delay < 2 & flights$dest == "BOS"
condicao2 <- flights$month == 1 & flights$sched_dep_time == 600
```

```
teste <- condicao1 | condicao2
```

```
flights[teste, ]
```

Capítulo 3 - Importando e exportando dados com o R

Exercício 1

3.1.A) Primeiro, sempre comece identificando o caractere especial que está separando cada coluna nesse arquivo. No caso do objeto `t` abaixo, esse caractere especial é o til (~). Com isso, podemos

utilizar o argumento `delim`, da função `read_delim()` para termos uma primeira leitura do arquivo, como demonstrado abaixo:

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

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

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

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

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

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

## # A tibble: 9 x 4
##   Setor          Produção  Receita `Gasto em P&D`
##   <chr>          <dbl>    <dbl>    <dbl>
## 1 Produtos alimentícios 1082837 19990755 335836
## 2 Bebidas              75953   2809321      NA
```

## 3 Produtos do fumo	6999	88635	12135
## 4 Produtos têxteis	415397	2580416	74683
## 5 Produtos de madeira	508878	1532069	27954
## 6 Celulose e outras pastas	2695	424519	2167
## 7 Refino de petróleo	7548	11431631	155073
## 8 Produtos químicos	317952	1335828	291409
## 9 Produtos farmacêuticos	62182	2497207	103873

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

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

```
## # A tibble: 9 x 4
```

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

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

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



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

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

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

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

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

```
## # A tibble: 9 x 4
##   Setor          Produção Receita `Gasto em P&D`
##   <chr>          <dbl>    <dbl> <chr>
## 1 Produtos alimentícios  10828. 199908. 3358,36
## 2 Bebidas                760.    28093. x
## 3 Produtos do fumo       70.0    8864. 121,35
## 4 Produtos têxteis      4154.   25804. 746,83
## 5 Produtos de madeira    5089.   15321. 279,54
## 6 Celulose e outras pastas 27.0    4245. 216,7
## 7 Refino de petróleo     75.5 114316. 1550,73
## 8 Produtos químicos     3180. 133583. 2914,09
## 9 Produtos farmacêuticos  622.    24972. 1038,73
```

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

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

## Warning: 1 parsing failure.
## row      col expected actual      file
##    2 Gasto em P&D a number      x literal data

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

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

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

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

```
998 1843,NA
999 1687,NA
1000 4569,NA
1001 4548,NA
1002 0.23837975086644292,2015-01-16
1003 0.41167997173033655,2018-05-18
1004 0.7460716762579978,2015-09-05
1005 0.723450553836301,2012-11-28
1006 0.614524137461558,2020-01-13
```

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

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

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

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

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

```
t <- "Data_execução*Unidades*Valor_compra
20/01/2020*21*R$ 3049,50
23/01/2020*502*R$ 1289,03
25/01/2020*90*R$ 678,00
02/02/2020*123*R$ 5401"
```

```
05/02/2020*45*R$ 1450,10
07/02/2020*67*R$ 2320,97
09/02/2020*187*R$ 6231,76"
```

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

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

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

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

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

```
## # A tibble: 7 x 3
##   Data_execução Unidades Valor_compra
```

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

Exercício 2

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

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

```
### Nomes de cada coluna
```

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

```
### Lembre-se que o caminho até o arquivo
```

```
### será diferente em sua máquina.
```

```
### Pois muito provavelmente você não possui um
```

```
### um usuário chamado Pedro.
```

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

```
## # A tibble: 853 x 16
```

```
##   Semestre      Ano Municipio      Cod_IBGE Area_2017 Area_2018 Area_Media
```

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

Capítulo 4 - Transformando dados com dplyr

Exercício 1

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

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

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

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

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

```
starwars %>%
  select(-all_of(vec)) %>%
  group_by(sex) %>%
  summarise(peso_medio = mean(mass, na.rm = TRUE))
```

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

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

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

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

```
## 4 Inglaterra      4  1.71
## 5 Japão          70 29.9
```

Exercício 2

4.2.A) Primeiro, identifique as principais colunas que são de seu interesse para responder a pergunta estipulada na questão. Para responder a pergunta, precisamos medir em quanto os preços cobrados por cada universidade aumentou e, para isso, não precisaremos das colunas `net_cost`, `income_lvl` e `campus`. Por isso, podemos rapidamente eliminar essas colunas com um `select()`.

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

custos

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

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

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

custos

## # A tibble: 40,991 x 4
##   name                                state total_price year
##   <chr>                                <chr>      <dbl> <dbl>
## 1 Piedmont International University NC      20174  2016
## 2 Piedmont International University NC      20514  2017
```



```
## 3 Piedmont International University NC      20829 2018
## 4 Piedmont International University NC      23000 2016
## 5 Piedmont International University NC      26430 2017
## 6 Piedmont International University NC      26870 2018
## 7 Kaplan University-Milwaukee WI           22413 2017
## 8 Kaplan University-Milwaukee WI           22492 2018
## 9 Kaplan University-Indianapolis IN         22413 2017
## 10 Kaplan University-Indianapolis IN        22492 2018
## # ... with 40,981 more rows
```

Lembre-se que, mesmo após aplicarmos `distinct()` sobre a base, pode haver dois `total_price`'s para uma mesma universidade em um mesmo ano. Ou seja, `distinct()` tratou de eliminar observações repetidas da tabela, isto é, observações que possuem exatamente os mesmos valores em todas as colunas. Sendo assim, podem existir na tabela, dois (ou mais) valores que se referem a uma mesma universidade e a um mesmo ano, mas que possuem valores diferentes na coluna `total_price`. O resultado dos comandos abaixo confirmam essa suspeita:

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

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

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

pois desejamos aplicar a média sobre `total_price`, dentro de cada ano (`year`) de cada universidade (`name`):

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

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

```
custos
```

```
## # A tibble: 30,066 x 3
## # Groups:   name [3,664]
##   name                year mean_price
##   <chr>              <dbl>     <dbl>
## 1 Aaniiih Nakoda College 2010      17030
## 2 Aaniiih Nakoda College 2011      17030
## 3 Aaniiih Nakoda College 2012      17030
## 4 Aaniiih Nakoda College 2013      17030
## 5 Aaniiih Nakoda College 2014      17030
## 6 Aaniiih Nakoda College 2015      17030
## 7 Aaniiih Nakoda College 2016      17030
## 8 Aaniiih Nakoda College 2017      17030
## 9 Aaniiih Nakoda College 2018      17030
## 10 Abilene Christian University 2011     38250
## # ... with 30,056 more rows
```

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

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

custos

## # A tibble: 30,066 x 4
##   name                year mean_price var_price
```

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

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

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

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

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

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

los_medanos
```

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

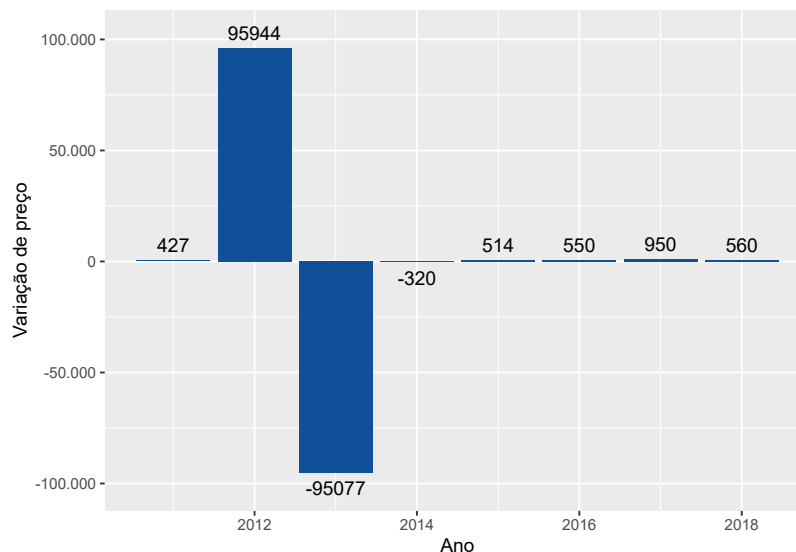
Expondo essa variação de maneira visual, temos:

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

```
los_medanos %>%
  ggplot(
    aes(x = year, y = var_price)
  ) +
  geom_col(
    fill = "#0f5099"
  ) +
  geom_text(
    aes(label = round(var_price, 0)),
    nudge_y = nudge
  ) +
  scale_y_continuous(
    labels = function(x){
      format(x, digits = 0, big.mark = ".")
    }
  ) +
  labs(
    x = "Ano",
    y = "Variação de preço"
  )
```

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

Exercício 3



Fonte: Elaboração própria do autor.

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

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

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

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

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

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

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

```
contagens
```

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

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

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

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

Exercício 4

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Capítulo 6 - Introdução a base de dados relacionais no R

Exercício 1

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

Por morarmos no Brasil, universidades americanas nos cobrariam o preço de um estudante não residente do estado de sua sede (*out of state*). O custo total para esse tipo de aluno está na coluna

out_of_state_total da tabela tuition_cost. Já o salário potencial de um aluno que acaba de se formar, é descrito na coluna early_career_pay da tabela salary_potential.

```
library(tidyverse)
```

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

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

```
##
## -- Column specification -----
## cols(
##   name = col_character(),
##   state = col_character(),
##   state_code = col_character(),
##   type = col_character(),
##   degree_length = col_character(),
##   room_and_board = col_double(),
##   in_state_tuition = col_double(),
##   in_state_total = col_double(),
##   out_of_state_tuition = col_double(),
##   out_of_state_total = col_double()
## )
```

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

```
##
## -- Column specification -----
## cols(
##   rank = col_double(),
##   name = col_character(),
##   state_name = col_character(),
##   early_career_pay = col_double(),
##   mid_career_pay = col_double(),
##   make_world_better_percent = col_double(),
##   stem_percent = col_double()
## )
```

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

```

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

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

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

custo

## # A tibble: 2,973 x 4
##   nome_universidade      custo_anual  anos custo_total
##   <chr>              <dbl> <dbl>      <dbl>
## 1 Aaniiih Nakoda College      2380     2        4760
## 2 Abilene Christian University  45200     4       180800
## 3 Abraham Baldwin Agricultural College  21024     2        42048
## 4 Academy College      17661     2        35322
## 5 Academy of Art University   44458     4       177832
## 6 Adams State University    29238     4       116952
## 7 Adelphi University      54690     4       218760
## 8 Adirondack Community College  21595     2        43190
## 9 Adrian College      48405     4       193620
## 10 Advanced Technology Institute  13680     2        27360
## # ... with 2,963 more rows

```

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

```

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

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

```

Com essas informações em mãos, podemos utilizar a função `inner_join()` para unir as duas tabelas criadas (`salario` e `custo`) em uma só. Em seguida, precisamos apenas dividir o custo total do

curso pela renda esperada para adquirir uma estimativa dos anos de trabalho necessários para repor o investimento aplicado no curso. Como você pode ver abaixo, um aluno formado na Adams State University levaria em torno de 2,63 anos (isto é, $2,63 \times 365 \approx 960$ dias) de trabalho para recompor os valores dispendidos em sua graduação.

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

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

Exercício 2

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

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

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

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

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

```
library(tidyverse)

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

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

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

## Joining, by = "Id_vendedor"

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

Exercício 3

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

Para corrigirmos esse problema, podemos: 1) renomear uma das colunas que representa a *key* do *join*, de modo que os seus nomes fiquem iguais; ou 2) dizer explicitamente à `left_join()`, quais são

as colunas equivalente entre essas tabelas, através do argumento `by` da função. Temos a capacidade de realizar a segunda opção de uma maneira bem direta, como demonstrado abaixo:

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

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

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

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

filmes %>%
  left_join(
    filmes_receita
  )

## Joining, by = "FilmeId"

## # A tibble: 14 x 8
##   FilmeId Titulo      Diretor      Ano DuracaoMinutos Nota_do_publico
##   <dbl> <chr>      <chr>      <dbl>      <dbl>      <dbl>
## 1      1 1 Toy Story    John Lasse~ 1995          81          83
```

```
## 2      2 A Bug's Life      John Lasse~ 1998      95      72
## 3      3 Toy Story 2      John Lasse~ 1999      93      79
## 4      4 Monsters, Inc.   Pete Docter 2001      92      81
## 5      5 Finding Nemo     Andrew Sta~ 2003     107     82
## 6      6 The Incredibles  Brad Bird  2004     116      8
## 7      7 Cars             John Lasse~ 2006     117     72
## 8      8 Ratatouille      Brad Bird  2007     115      8
## 9      9 WALL-E          Andrew Sta~ 2008     104     85
## 10     10 Up              Pete Docter 2009     101     83
## 11     11 Toy Story 3     Lee Unkrich 2010     103     84
## 12     12 Cars 2         John Lasse~ 2011     120     64
## 13     13 Brave          Brenda Cha~ 2012     102     72
## 14     14 Monsters Unive~ Dan Scanlon 2013     110     74
## # ... with 2 more variables: Receita_interna <dbl>,
## #   Receita_internacional <dbl>
```

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

Exercício 1

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

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

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

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

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

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

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

```
nrow(world_bank_pop)
```

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

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

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

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

tab_tidy

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

```
## 10 ABW      2009      44052      -0.731      101453      0.0986
## # ... with 4,742 more rows
```

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

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

```
pop_total
```

```
## # A tibble: 264 x 20
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2006`
##   <chr>    <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 ABW      SP.POP.TO~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e4 1.00e5 1.01e5
## 2 AFG      SP.POP.TO~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e7 2.51e7 2.59e7
## 3 AGO      SP.POP.TO~ 1.64e7 1.70e7 1.76e7 1.82e7 1.89e7 1.96e7 2.03e7
## 4 ALB      SP.POP.TO~ 3.09e6 3.06e6 3.05e6 3.04e6 3.03e6 3.01e6 2.99e6
## 5 AND      SP.POP.TO~ 6.54e4 6.73e4 7.00e4 7.32e4 7.62e4 7.89e4 8.10e4
## 6 ARB      SP.POP.TO~ 2.84e8 2.90e8 2.96e8 3.02e8 3.09e8 3.16e8 3.24e8
## 7 ARE      SP.POP.TO~ 3.15e6 3.33e6 3.51e6 3.74e6 4.09e6 4.58e6 5.24e6
## 8 ARG      SP.POP.TO~ 3.71e7 3.75e7 3.79e7 3.83e7 3.87e7 3.91e7 3.96e7
## 9 ARM      SP.POP.TO~ 3.07e6 3.05e6 3.03e6 3.02e6 3.00e6 2.98e6 2.96e6
## 10 ASM     SP.POP.TO~ 5.75e4 5.82e4 5.87e4 5.91e4 5.93e4 5.91e4 5.86e4
## # ... with 254 more rows, and 11 more variables: 2007 <dbl>, 2008 <dbl>,
## #   2009 <dbl>, 2010 <dbl>, 2011 <dbl>, 2012 <dbl>, 2013 <dbl>,
## #   2014 <dbl>, 2015 <dbl>, 2016 <dbl>, 2017 <dbl>
```

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

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



```

group_by(country) %>%
mutate(
  variacao = valor - lag(valor)
)

pop_total

## # A tibble: 4,752 x 5
## # Groups:   country [264]
##   country indicator  ano    valor variacao
##   <chr>    <chr>    <chr> <dbl>    <dbl>
## 1 ABW      SP.POP.TOTL 2000   90853      NA
## 2 ABW      SP.POP.TOTL 2001   92898    2045
## 3 ABW      SP.POP.TOTL 2002   94992    2094
## 4 ABW      SP.POP.TOTL 2003   97017    2025
## 5 ABW      SP.POP.TOTL 2004   98737    1720
## 6 ABW      SP.POP.TOTL 2005  100031    1294
## 7 ABW      SP.POP.TOTL 2006  100832     801
## 8 ABW      SP.POP.TOTL 2007  101220     388
## 9 ABW      SP.POP.TOTL 2008  101353     133
## 10 ABW     SP.POP.TOTL 2009  101453     100
## # ... with 4,742 more rows

```

Capítulo 8 - Visualização de dados com ggplot2

Exercício 1

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

```

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

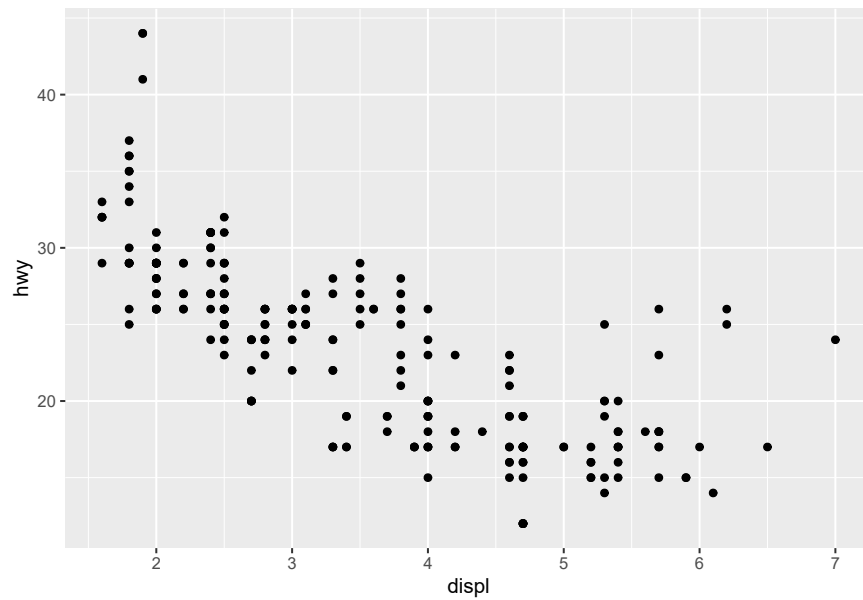
```

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

```

ggplot(data = diamonds) +
  geom_point(

```

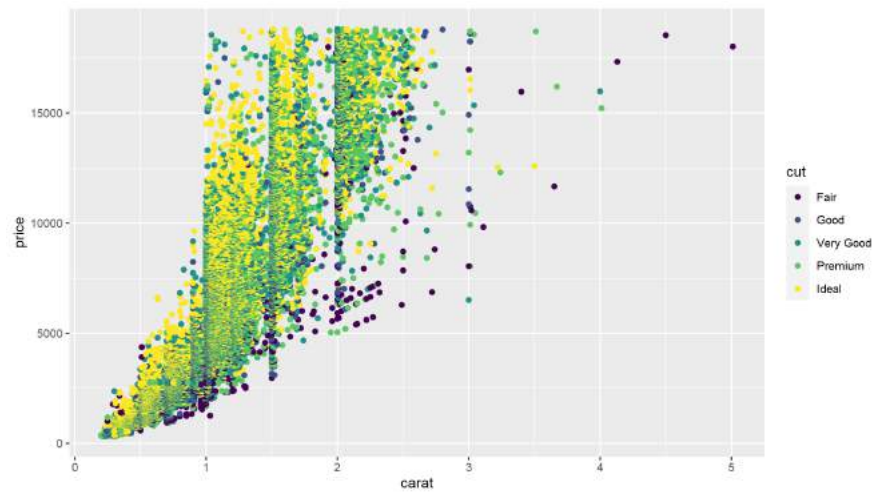


Fonte: Elaboração própria do autor.

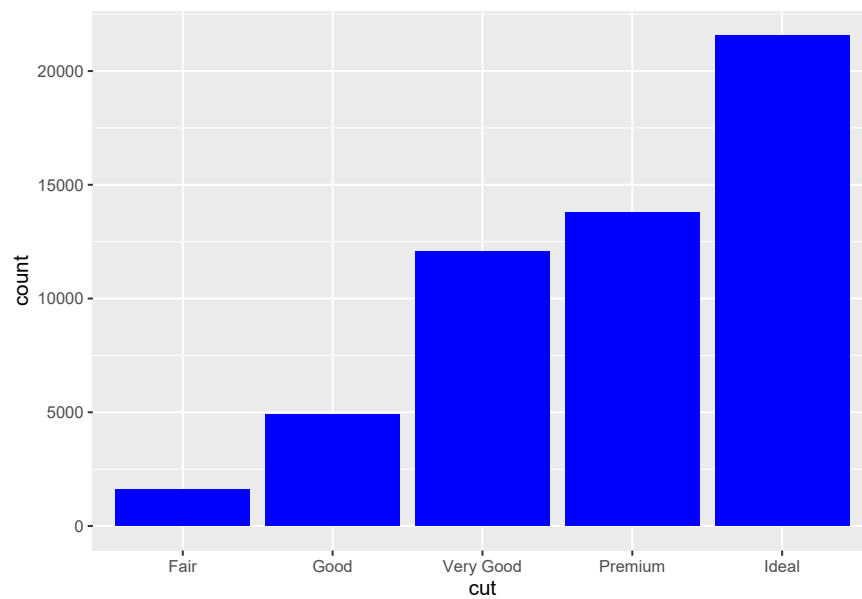
```
aes(x = carat, y = price, color = cut)
)
```

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

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



Fonte: Elaboração própria do autor.



Fonte: Elaboração própria do autor.

Exercício 2

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

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

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

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

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

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



Fonte: Elaboração própria do autor.

Esse é certamente um método simples e eficaz de desenhar essa bandeira. Entretanto, uma outra forma de desenharmos essa bandeira, seria desenhando 6 retângulos, um sobre o outro. Podemos realizar esse processo por meio de `geom_rect()`. Todos os retângulos possuem a mesma largura (de 10 unidades), porém, as alturas se reduzem em 1 unidade (6, 5, 4, 3,...) em cada retângulo. Ou seja, o primeiro retângulo (de cor vermelha) é o maior de todos, ou, dito de outra forma, sua altura cobre todas as 6 faixas da bandeira. Já o último retângulo (de cor roxa), é o menor de todos, pois sua altura cobre apenas 1 única faixa.

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

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

```
scale_fill_identity() +  
theme_void()
```

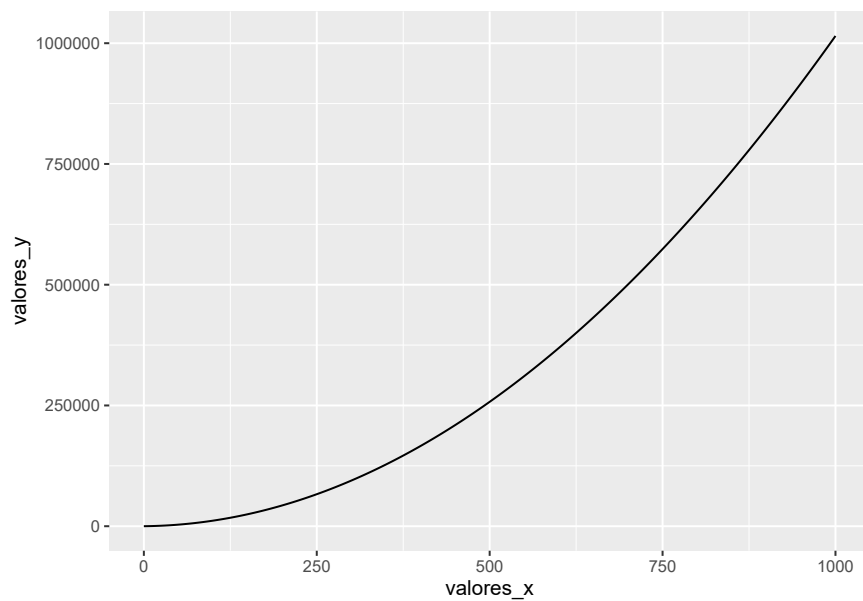


Fonte: Elaboração própria do autor.

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

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

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

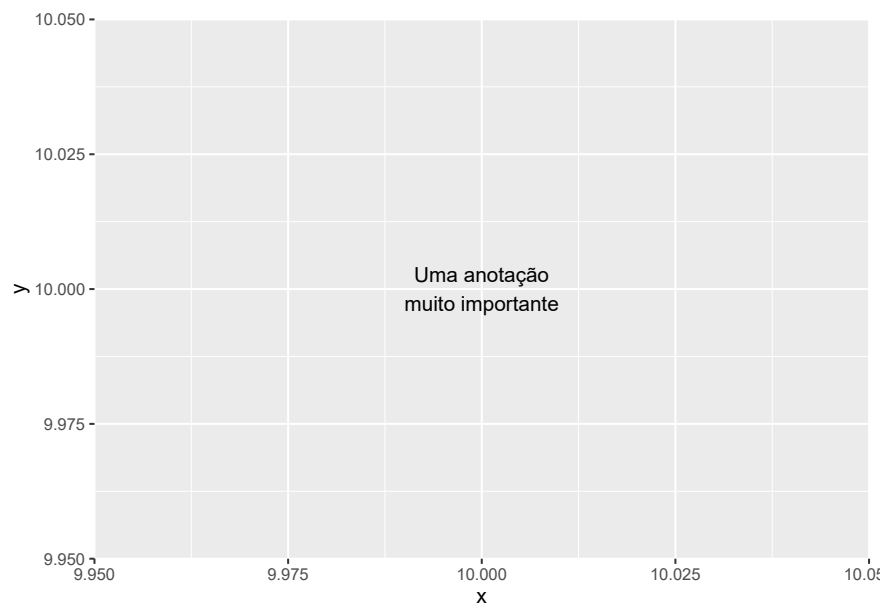


Fonte: Elaboração própria do autor.

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

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

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



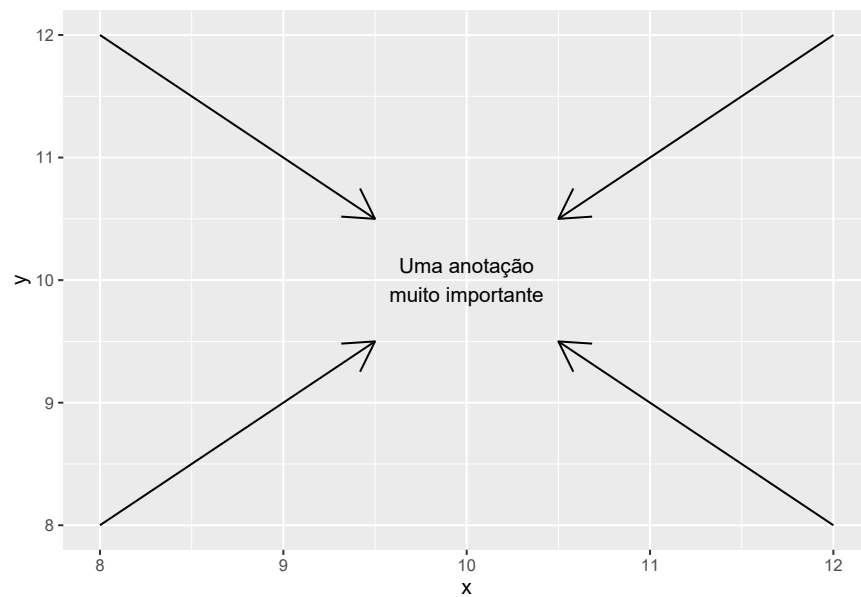
Fonte: Elaboração própria do autor.

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

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



```
ggplot() +  
  geom_text(  
    aes(x = 10, y = 10, label = anotacao)  
  ) +  
  geom_segment(  
    aes(x = x, y = y, xend = xend, yend = yend,  
        group = id),  
    data = setas,  
    ### Ao definirmos o argumento arrow  
    ### com a função arrow(),  
    ### podemos adicionar a cabeça da  
    ### seta de forma fácil e automática  
    arrow = arrow()  
  )
```

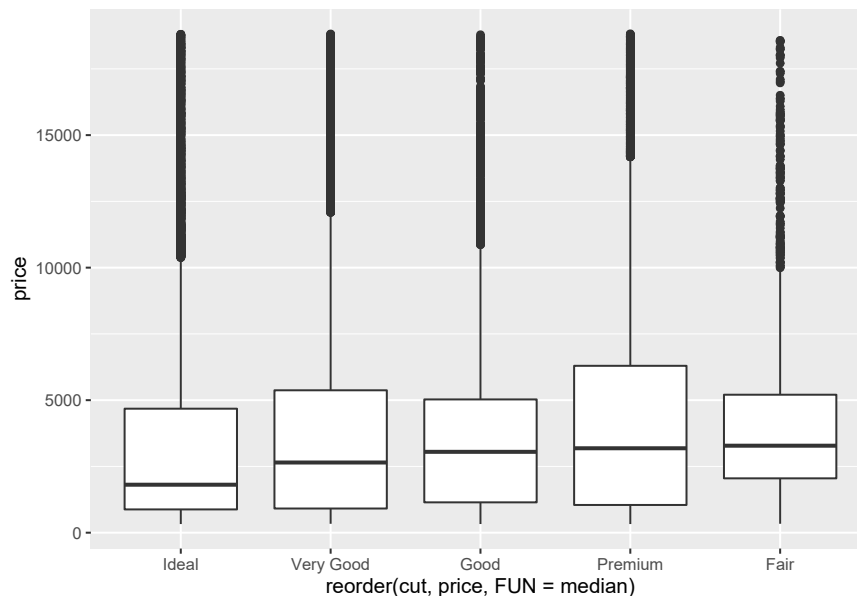


Fonte: Elaboração própria do autor.

Exercício 3

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

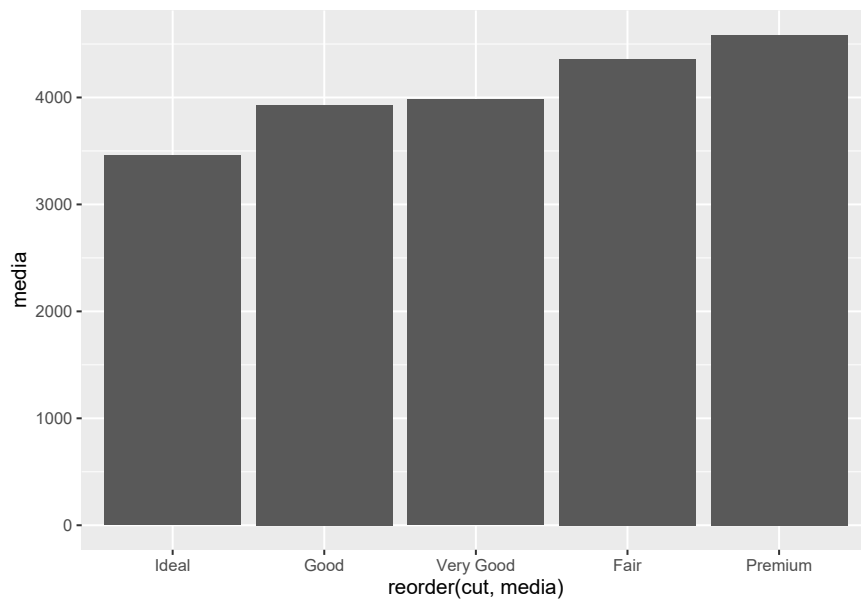
```
diamonds %>%
  ggplot() +
  geom_boxplot(
    aes(
      x = reorder(cut, price, FUN = median),
      y = price
    )
  )
```



Fonte: Elaboração própria do autor.

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

```
estatisticas <- diamonds %>%  
  group_by(cut) %>%  
  summarise(  
    media = mean(price, na.rm = TRUE)  
  )  
  
estatisticas %>%  
  ggplot() +  
  geom_col(  
    aes(  
      x = reorder(cut, media),  
      y = media  
    )  
  )
```



Fonte: Elaboração própria do autor.

Capítulo 9 - Configurando componentes estéticos do gráfico no ggplot2

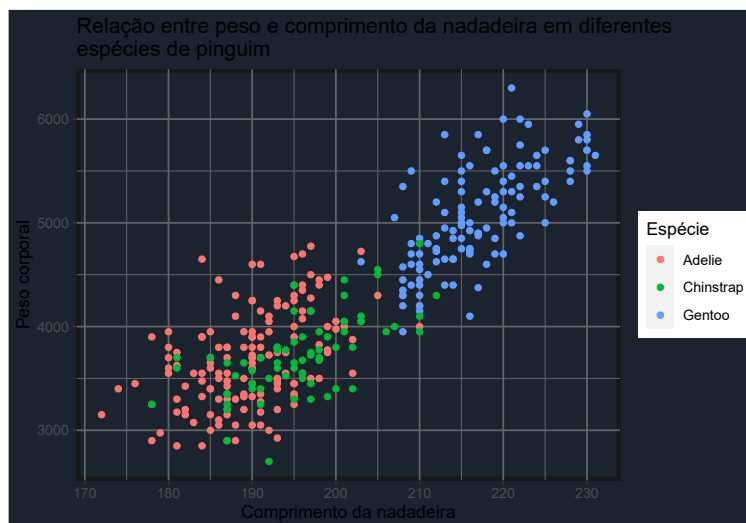
Exercício 1

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

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

```
plot_exemplo + tema
```

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



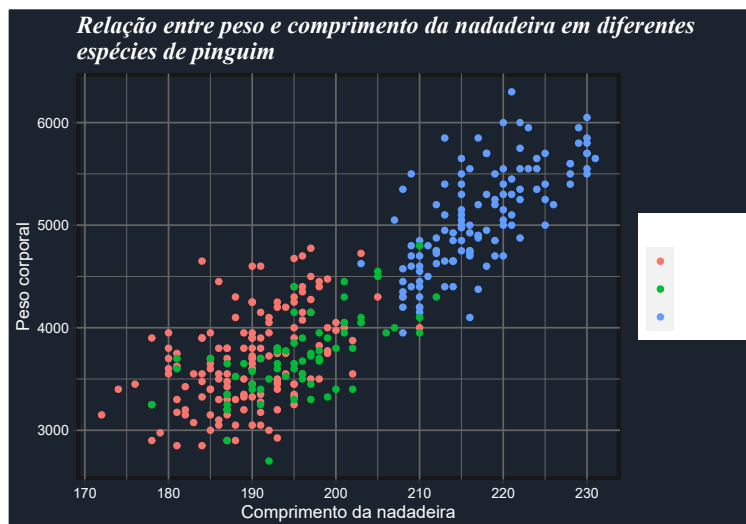
Fonte: Elaboração própria do autor.

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

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

```
plot_exemplo + tema
```

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



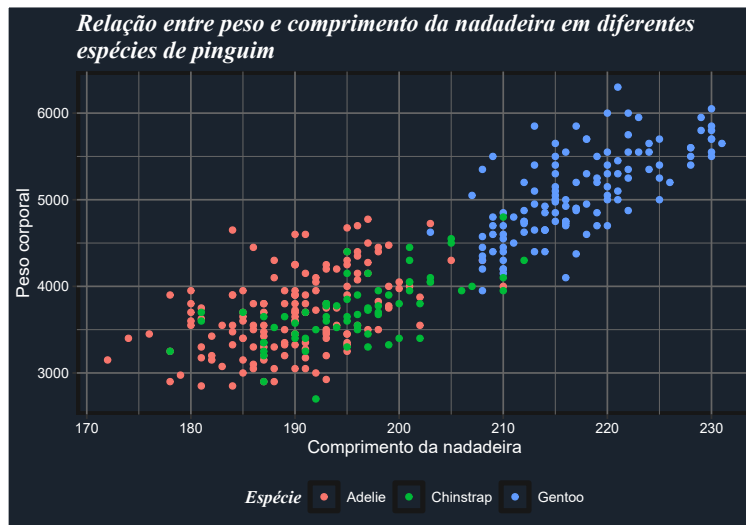
Fonte: Elaboração própria do autor.

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

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

```
plot_exemplo + tema
```

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

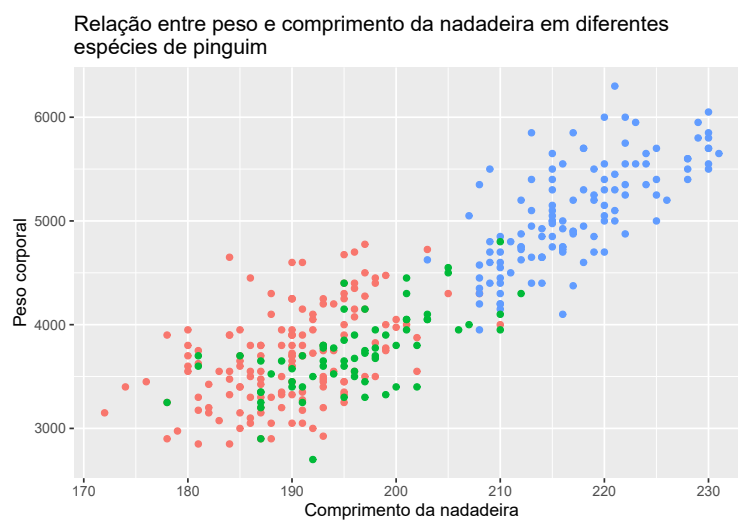


Fonte: Elaboração própria do autor.

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

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

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

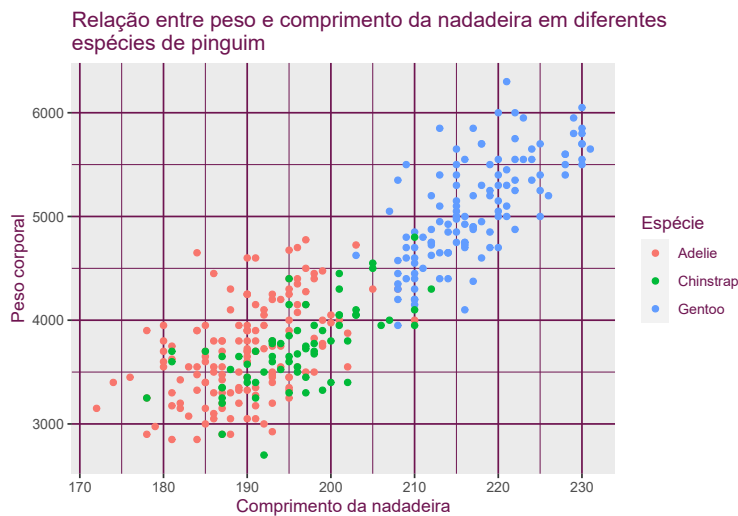


Fonte: Elaboração própria do autor.

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

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

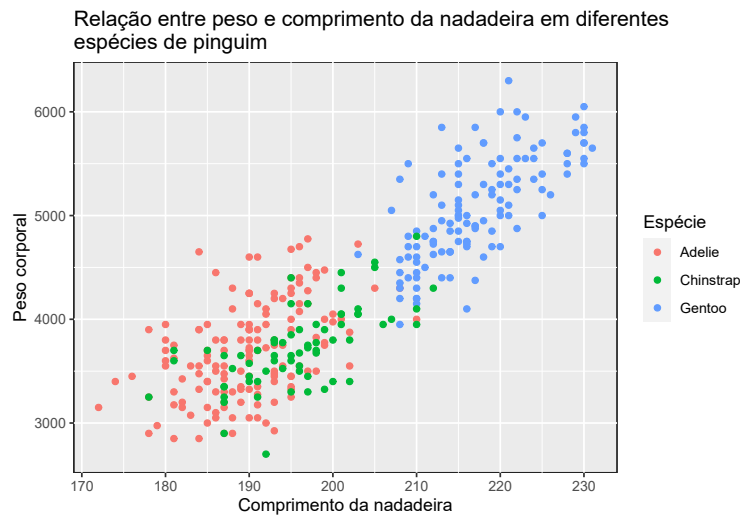
Warning: Removed 2 rows containing missing values (geom_point).



Fonte: Elaboração própria do autor.

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

```
plot_exemplo +  
  theme(panel.background = element_rect(color = "#222222"))  
  
## Warning: Removed 2 rows containing missing values (geom_point).
```



Fonte: Elaboração própria do autor.

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

Exercício 1

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

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

words[index]

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

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

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

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

10.1.C) A primeira sequência citada (“s-p-a-c-e”) é bem específica e, por isso, talvez seja mais fácil utilizarmos um *metacharacter* de alternância | para separar essa sequência como um caso especial. Do outro lado do *metacharacter* | podemos a expressão "ess\$", ou ainda, a expressão "e(s{2})\$", ambas são expressões equivalentes.

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

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

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

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

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

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

print(resultado, max = 30)

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

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

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

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

quase todas as palavras desse vetor contém uma sequência de 3 letras quaisquer em algum lugar dentro de si.

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

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

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

Exercício 2

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

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

```
library(tidyverse)

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

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

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

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

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

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

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

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

Todavia, por algum motivo, os dia abaixo de 10 (ou seja, dias 1, 2, 3, 4, 5, etc.) estão desacompanhados do zero inicial. Ou seja, uma data que deveria estar escrita como “04/05/2020” está na verdade escrita na forma “4/05/2020”. Portanto, uma alternativa para corrigir esse problema é incluir o *meta-character* da dúvida (?) no zero, formando assim a expressão "[0-9]?[0-9]/[0-9]{2}/[0-9]{4} [0-9]{2}:[0-9]{2}:[0-9]{2} -03". Dessa forma, estamos definindo indiretamente que o dia pode ter 1 ou 2 caracteres, pois o primeiro caractere nessa região é opcional.

Caso você queira ser mais estrito, ou, mais preciso a respeito das datas, você poderia utilizar a expressão "(3[0-1]|[0-2][0-9])/([0-1-9]|1[1-2])/2[0-9]{3}". O mesmo poderia ser feito

com o horário, o qual seria melhor representado por uma expressão como "(0[0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9])". Entranto, se tratando dos *strings* contidos no vetor *compras*, nós não precisamos ser tão específicos assim.

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

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

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

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

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

## [1] "[a-zA-ZÁÉÍÓÚáéíóú+)([0-9]{3}[.][0-9]{3}[.][0-9]{2}~"
```

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

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

```
tab <- resultado %>%
  read_csv2(col_names = FALSE)
```

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

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

parte <- str_extract(
  compras, pattern = horario
)

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

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

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

print(contagens)
```

```
## Dia 26 Dia 12 Dia 14 Dia 20 Dia 3 Dia 13 Dia 19 Dia 29 Dia 7 Dia 24
##      56      49      46      46      42      42      42      41      40      39
## Dia 4  Dia 9 Dia 10 Dia 27 Dia 1 Dia 21 Dia 22 Dia 8 Dia 25 Dia 6
##      38      38      38      38      37      37      37      36      36      35
```

```
## Dia 16 Dia 30 Dia 11 Dia 18 Dia 5 Dia 15 Dia 17 Dia 2 Dia 23 Dia 28
##      35      35      34      33      32      32      32      29      29      28
## Dia 31
##      18
```

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

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

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

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

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

A resposta para essas perguntas se baseia no princípio de que pesquisas por expressões regulares ocorrem da esquerda para a direita em um *string* (FRIEDL, 2006). Portanto, a expressão `"[0-9]{3}"` é capaz de extrair os 3 primeiros dígitos de cada CPF, pelo simples fato de que o CPF é o primeiro campo numérico a aparecer no *string*, antes do código de identificação do produto, e antes da data e horário da compra.

Capítulo 12 - Introdução à variáveis de tempo com lubridate

Exercício 1

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


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

```
as.Date(vec)
```

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

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

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

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

```
dmy(vec)
```

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

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

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

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

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

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

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

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

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

Exercício 2

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

```
numero_no_excel <- c(
  43881.1527777778,
  43881.15625,
  43881.15972222226,
  43881.1632060185,
  43881.1666666667
)
```

Por isso, ao fornecermos esses valores para a função `as.POSIXct()` precisamos converter esses valores para segundos (ao multiplicá-los pelo fator $60 \times 60 \times 24$). Como foi destacado no enunciado, o Excel utiliza o dia 30 de dezembro de 1899 como o seu ponto de origem e, por isso, precisamos fornecer esse dia ao argumento `origin`. Dessa maneira, `as.POSIXct()` vai utilizar a data 30 de dezembro de 1899 (ao invés de 01 de janeiro de 1970) como base para calcular as datas.

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

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

print(ajuste)
```

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

Apêndice A

PNAD Contínua: arquivo CSV para input

Neste apêndice, você pode encontrar logo abaixo, os comandos necessários para importar um arquivo CSV para o seu R. Esse arquivo CSV, possui as especificações de cada coluna presente nos arquivos dos microdados da PNAD Contínua. Sendo que essas especificações, são necessárias para importar os microdados da PNAD Contínua em qualquer programa estatístico.

Esse arquivo CSV foi construído no dia 26/11/2020, com base nas especificações das colunas presentes no arquivo *input* (input_PNADC_trimestral.txt), que você pode encontrar na [página do servidor](#) em que os microdados da PNAD Contínua são hospedados, ou então, você pode baixar um ZIP (Dicionario_input.zip) contendo este arquivo input, através deste [link](#). Este arquivo CSV, foi utilizado para importar os microdados da PNAD Contínua referente ao 1º trimestre de 2020, na seção **Um estudo de caso: lendo os microdados da PNAD Contínua com read_fwf()** desta obra.

```
library(readr)

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

input <- read_csv(
  paste0(github, pasta, arquivo),
  col_names = c("nome_coluna", "largura", "coluna_numerica")
)
```



Referências Bibliográficas

ADLER, J. *R in a Nutshell*. Sebastopol, CA: O'Reilly, 2010. ISBN 978-0-596-80170-0.

BRAGA, D.; ASSUNCAO, G.; HIDALGO, L. Pnadcibge: Downloading, reading and analysing pnadc microdata. In: . CRAN R Package, 2020. Disponível em: <<https://cran.r-project.org/web/packages/PNADcIBGE/index.html>>.

CÂMARA, G.; MONTEIRO, A. M. V. Conceitos básicos em ciência da geoinformação. In: CÂMARA, G.; DAVIS, C.; MONTEIRO, A. M. V. (Org.). *Introdução à Ciência da Geoinformação*. São José dos Campos: Instituto Nacional de Pesquisas Espaciais - INPE, 2001. p. 7–42. Disponível em: <<http://www.dpi.inpe.br/gilberto/livro/introd/>>.

CHAMBERS, J. M. *Software for Data Analysis: Programming with r*. New York, NY: Springer, 2008. ISBN 978-0-387-75935-7.

CHAMBERS, J. M. *Extending R*. Boca Raton, FL: CRC Press, 2016. (The R Series). ISBN 978-1-4987-7572-4.

CHANG, W. *R Graphics Cookbook*. Sebastopol, CA: O' Reilly Media, Inc., 2012. ISBN 978-1-449-31695-2. Disponível em: <<https://r-graphics.org>>.

CHASE, W. *Custom fonts and plot quality with ggplot on Windows*. 2019. Disponível em: <<https://www.williamrchase.com/post/custom-fonts-and-plot-quality-with-ggplot-on-windows/>>. Acesso em: 21 nov. de 2020.

CHASE, W. *Constructing a Career in Dataviz: the how*. 2020. Disponível em: <<https://www.williamrchase.com/post/constructing-a-career-in-dataviz-the-how/>>. Acesso em: 06 dez. de 2020.

FRERY, A. C.; PERCIANO, T. *Introduction to Image Processing Using R: Learning by examples*. 3. ed. Londres: Springer, 2013. ISBN 978-1-4471-4949-1.

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

GILLESPIE, C.; LOVELACE, R. *Efficient R Programming*. Sebastopol, CA: O' Reilly Media, Inc., 2017. ISBN 978-1491950784. Disponível em: <<https://csgillespie.github.io/efficientR/>>.

- GROLEMUND, G. *Hands-On Programming with R*. Sebastopol, CA: O’ Reilly Media, Inc., 2014. Disponível em: <<https://rstudio-education.github.io/hopr/>>.
- GROTHENDIECK, G.; PETZOLDT, T. Date and time classes in r. *R News*, v. 4, n. 1, p. 29–31, 2004. ISSN 1609-3631.
- HARALAMBOUS, Y. *Fonts & Encodings*. Sebastopol, CA: O’ Reilly Media, Inc., 2007. ISBN 978-0-596-10242-5.
- HUGHES, J. F. et al. *Computer Graphics: Principles and practice*. 3. ed. [S.l.]: Addison-Wesley, 2014. ISBN 978-0-321-39952-6.
- IHAKA, R.; GENTLEMAN, R. R. A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, v. 5, n. 3, p. 299–314, 1996.
- INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. *Pesquisa Nacional por Amostra de Domicílios Contínua*: Notas técnicas. Rio de Janeiro, 2019. Disponível em: <https://biblioteca.ibge.gov.br/visualizacao/livros/liv101674_notas_tecnicas.pdf>.
- LONG, J. D.; TEETOR, P. *R Cookbook*. 2nd. ed. Sebastopol, CA: O’ Reilly Media, Inc., 2019. Disponível em: <<https://rc2e.com>>.
- MCDONNELL, R. M.; OLIVEIRA, E.; GIANNOTTI, R. cepr: Busca ceprs brasileiros. In: . CRAN R Package, 2020. Disponível em: <<https://cran.r-project.org/web/packages/cepr/index.html>>.
- MURRAY, J. D.; VANRYPER, W. *Encyclopedia of Graphics Files Formats*. 2. ed. Sebastopol, CA: O’Reilly Media, 1996. ISBN 1-56592-161-5.
- MURRELL, P. *R Graphics*. 1. ed. Boca Raton, FL: Chapman & Hall - CRC Press, 2006. ISBN 1-58488-486-X.
- NIELD, T. *Getting Started with SQL: A hands-on approach for beginners*. 1. ed. Sebastopol, CA: O’Reilly Media, 2016. ISBN 978-1-491-93861-4.
- PEDERSEN, T. L. *Updates to ragg and systemfonts*. 2020. Disponível em: <<https://www.tidyverse.org/blog/2020/05/updates-to-agg-and-systemfonts/>>. Acesso em: 24 nov. de 2020.
- PENG, R. D. *R Programming for Data Science*. Leanpub, 2015. Disponível em: <<https://bookdown.org/rdpeng/rprogdatascience/>>.
- PEREIRA, R. H. et al. geobr: Loads shapefiles of official spatial data sets of brazil. In: IPEA - INSTITUTO DE PESQUISA ECONÔMICA APLICADA. CRAN R Package, 2020. Disponível em: <<https://cran.r-project.org/web/packages/geobr/index.html>>.
- PETRUZALEK, D. read.dbc: Read data stored in dbc (compressed dbf) files. In: . CRAN R Package, 2016. Disponível em: <<https://cran.r-project.org/web/packages/read.dbc/index.html>>.
- QIU, Y. showtext: Using system fonts in r graphics. *The R Journal*, v. 7, n. 1, p. 99–108, 2015. Disponível em: <<https://doi.org/10.32614/RJ-2015-008>>.

R CORE TEAM. *An Introduction to R: A programming environment for data analysis and graphics*. Version 4.0.3. [S.l.], 2020. Disponível em: <https://cran.r-project.org/doc/manuals/r-release/R-intro.html>.

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

RIPLEY, B. D.; HORNIK, K. Date-time classes. *R News*, v. 1, n. 2, p. 8–11, 2001. ISSN 1609-3631.

SIQUEIRA, R. P. sidrar: An interface to ibge's sidra api. In: . CRAN R Package, 2020. Disponível em: <https://cran.r-project.org/web/packages/sidrar/index.html>.

WICKHAM, H. Tidy data. *The Journal of Statistical Software*, v. 59, 2014. Disponível em: <http://www.jstatsoft.org/v59/i10/>.

WICKHAM, H. *Advanced R*. 2. ed. Boca Raton, Florida: CRC Press, 2015. Disponível em: <https://adv-r.hadley.nz>.

WICKHAM, H. *R Packages*. Sebastopol, CA: O' Reilly Media, Inc., 2015. Disponível em: <http://r-pkgs.had.co.nz>.

WICKHAM, H. *ggplot2: Elegant graphics for data analysis*. 2. ed. Springer International Publishing, 2016. (Use R!). ISBN 978-3-319-24275-0. Disponível em: <https://ggplot2-book.org>.

WICKHAM, H. *dplyr 1.0.0 available now!* 2020. Disponível em: <https://www.tidyverse.org/blog/2020/06/dplyr-1-0-0/>. Acesso em: 29 dez. de 2020.

WICKHAM, H.; GROLEMUND, G. *R for Data Science*. Sebastopol, CA: O' Reilly Media, Inc., 2017. Disponível em: <https://r4ds.had.co.nz>.

WILKINSON, L. *The Grammar of Graphics*. 2. ed. Verlag, NY: Springer, 2005. ISBN 978-0-387-28695-2.
