

# Análise de Dados Financeiros e Econômicos com o R - Versão Online

Marcelo S. Perlin ([marcelo.perlin@ufrgs.br](mailto:marcelo.perlin@ufrgs.br))

2022-11-23

## **Análise de Dados Financeiros e Econômicos com o R**

por **Marcelo Scherer Perlin**

© 2021 Marcelo S. Perlin. All rights reserved.

Publicação Independente. Impresso sob demanda por Amazon.com.  
Versão Online disponível em <https://www.msperlin.com/adfeR/>

**Revisão de texto:** Diversos  
**Capa:** Rubens Lima - <http://capista.com.br/>  
**ISBN (paperback):** 9798707107559  
**ISBN (hardcover):** 9798711081821

### **Histórico de edições:**

Primeira edição: 2017-03-01  
Segunda edição: 2018-07-01  
Terceira edição: 2021-03-01

Embora o autor tenha boa fé para garantir que as instruções e o código contidos neste trabalho sejam precisos, ele se exime de toda responsabilidade por erros ou omissões, incluindo, sem limitação, a responsabilidade por danos resultantes do uso ou da confiança neste trabalho e em seus resultados. O uso das informações contidas neste trabalho é por sua conta e risco. Se qualquer código deste livro estiver sujeito a licenças de código aberto ou direitos de propriedade intelectual de terceiros, o cumprimento desses direitos e licenças é de sua responsabilidade como usuário.

Todo o código contido no livro é disponibilizado pela generosa licença do MIT. Portanto, sinta-se livre para utilizá-lo no seu trabalho, desde de que a origem do código seja citada. Uma sugestão de citação é disponibilizada abaixo:

Perlin, M. S. *Análise de Dados Financeiros com o R*. Terceira Edição, Porto Alegre: Marcelo S. Perlin (publicação independente), 2021.

# Sumário

<b>Sobre Nova Edição</b>	<b>9</b>
<b>Prefácio</b>	<b>11</b>
Material Suplementar . . . . .	13
Conteúdo para Instrutores . . . . .	14
<b>1 Introdução</b>	<b>15</b>
1.1 O que é o R . . . . .	16
1.2 Por que Escolher o R . . . . .	16
1.3 Usos do R . . . . .	17
1.4 Como Instalar o R . . . . .	18
1.5 Instalando o RStudio . . . . .	22
1.6 Recursos na Internet . . . . .	23
1.7 Organização e Material do Livro . . . . .	23
1.8 Exercícios . . . . .	25
<b>2 Operações Básicas no R</b>	<b>29</b>
2.1 Como o R Funciona? . . . . .	29
2.2 Objetos e Funções . . . . .	31
2.3 O Formato Brasileiro . . . . .	32
2.4 Tipos de Arquivos . . . . .	33
2.5 Explicando a Tela do RStudio . . . . .	34
2.6 Pacotes do R . . . . .	37
2.6.1 Instalando Pacotes do CRAN . . . . .	39
2.6.2 Instalando Pacotes do Github . . . . .	40
2.6.3 Carregando Pacotes . . . . .	41
2.6.4 Atualizando Pacotes . . . . .	42
2.7 Executando Códigos em um <i>Script</i> . . . . .	44

2.8	Testando Código . . . . .	46
2.9	Criando Objetos Simples . . . . .	47
2.10	Criando Vetores . . . . .	49
2.11	Conhecendo os Objetos Criados . . . . .	50
2.12	Mostrando e Formatando Informações na Tela . . . . .	52
2.13	Conhecendo o Tamanho dos Objetos . . . . .	56
2.14	Selecionando Elementos de um Vetor Atômico . . . . .	59
2.15	Limpando a Memória . . . . .	62
2.16	Mostrando e Mudando o Diretório de Trabalho . . . . .	63
2.17	Comentários no Código . . . . .	66
2.18	Cancelando a Execução de um Código . . . . .	68
2.19	Procurando Ajuda . . . . .	68
2.20	Utilizando <i>Code Completion</i> com a Tecla <i>tab</i> . . . . .	70
2.21	Interagindo com Arquivos e o Sistema Operacional . . . . .	73
2.21.1	Listando Arquivos e Pastas . . . . .	73
2.21.2	Apagando Arquivos e Diretórios . . . . .	75
2.21.3	Utilizando Arquivos e Diretórios Temporários . . . . .	76
2.21.4	Baixando Arquivos da Internet . . . . .	77
2.22	Exercícios . . . . .	78
<b>3</b>	<b>Desenvolvendo Rotinas de Pesquisa</b>	<b>81</b>
3.1	Etapas de uma Pesquisa . . . . .	81
3.2	A Estrutura de Diretórios . . . . .	83
3.3	Pontos Importantes em uma Pesquisa . . . . .	85
3.4	Exercícios . . . . .	87
<b>4</b>	<b>Importação e Exportação de Dados Locais</b>	<b>89</b>
4.1	Pacote <i>adfeR</i> . . . . .	90
4.2	Arquivos <i>csv</i> . . . . .	91
4.2.1	Importação de Dados . . . . .	94
4.2.2	Exportação de Dados . . . . .	97
4.3	Arquivos <i>Excel</i> ( <i>xls</i> e <i>xlsx</i> ) . . . . .	98
4.3.1	Importação de Dados . . . . .	99
4.3.2	Exportação de Dados . . . . .	100
4.4	Formato <i>.RData</i> e <i>.rds</i> . . . . .	103
4.4.1	Importação de Dados . . . . .	104
4.4.2	Exportação de Dados . . . . .	104
4.5	Arquivos <i>fst</i> (pacote <i>fst</i> ) . . . . .	105
4.5.1	Importação de Dados . . . . .	106
4.5.2	Exportação de Dados . . . . .	106
4.5.3	Testando o Tempo de Execução do Formato <i>fst</i> . . . . .	106
4.6	Arquivos <i>SQLite</i> . . . . .	108
4.6.1	Importação de Dados . . . . .	109

4.6.2	Exportação de Dados . . . . .	110
4.7	Dados Não-Estruturados e Outros Formatos . . . . .	111
4.7.1	Exportando de Dados Não-Estruturados . . . . .	113
4.8	Selecionando o Formato . . . . .	114
4.9	Exercícios . . . . .	114
<b>5</b>	<b>Importação de Dados via Pacotes</b>	<b>117</b>
5.1	Pacote <code>GetQuandlData</code> . . . . .	118
5.1.1	Importando Múltiplas Séries . . . . .	121
5.2	Pacote <code>BatchGetSymbols</code> . . . . .	124
5.2.1	Baixando Dados da Composição do Ibovespa . . . . .	127
5.3	Pacote <code>GetTDData</code> . . . . .	128
5.4	Pacote <code>GetBCBData</code> . . . . .	133
5.5	Pacote <code>GetDFPData2</code> . . . . .	137
5.6	Pacote <code>GetFREData</code> . . . . .	141
5.7	Outros Pacotes . . . . .	143
5.7.1	Pacotes de Acesso Gratuito . . . . .	143
5.7.2	Pacotes Comerciais . . . . .	144
5.8	Acessando Dados de Páginas na Internet ( <i>Webscraping</i> ) . . . . .	144
5.8.1	Raspando Dados do Wikipedia . . . . .	144
5.9	Exercícios . . . . .	147
<b>6</b>	<b>Dataframes e outros Objetos</b>	<b>151</b>
6.1	Dataframes . . . . .	152
6.1.1	Criando dataframes . . . . .	152
6.1.2	Inspecionando um dataframe . . . . .	154
6.1.3	Operador de <i>pipeline</i> ( <code>%&gt;%</code> ) . . . . .	156
6.1.4	Acessando Colunas . . . . .	157
6.1.5	Modificando um dataframe . . . . .	159
6.1.6	Filtrando um dataframe . . . . .	161
6.1.7	Ordenando um dataframe . . . . .	162
6.1.8	Combinando e Agregando dataframes . . . . .	164
6.1.9	Extensões ao dataframe . . . . .	168
6.1.10	Outras Funções Úteis . . . . .	170
6.2	Listas . . . . .	172
6.2.1	Criando Listas . . . . .	173
6.2.2	Acessando os Elementos de uma Lista . . . . .	175
6.2.3	Adicionando e Removendo Elementos de uma Lista . . . . .	177
6.2.4	Processando os Elementos de uma Lista . . . . .	179
6.2.5	Outras Funções Úteis . . . . .	180
6.3	Matrizes . . . . .	182
6.3.1	Selecionando Valores de uma Matriz . . . . .	184
6.3.2	Outras Funções Úteis . . . . .	186

6.4	Exercícios . . . . .	187
<b>7</b>	<b>As Classes Básicas de Objetos</b>	<b>189</b>
7.1	Objetos Numéricos . . . . .	189
7.1.1	Criando e Manipulando Vetores Numéricos . . . . .	190
7.1.2	Acessando Elementos de um Vetor Numérico . . . . .	198
7.1.3	Modificando e Removendo Elementos de um Vetor Numérico . . . . .	199
7.1.4	Criando Grupos . . . . .	201
7.1.5	Outras Funções Úteis . . . . .	202
7.2	Classe de Caracteres (texto) . . . . .	205
7.2.1	Criando um Objeto Simples de Caracteres . . . . .	205
7.2.2	Criando Objetos Estruturados de Texto . . . . .	205
7.2.3	Objetos Constantes de Texto . . . . .	207
7.2.4	Selecionando Pedacos de um Texto . . . . .	208
7.2.5	Localizando e Substituindo Pedacos de um Texto . . . . .	209
7.2.6	Separando Textos . . . . .	212
7.2.7	Descobrimdo o Número de Caracteres de um Texto . . . . .	213
7.2.8	Gerando Combinações de Texto . . . . .	213
7.2.9	Codificação de Objetos <code>character</code> . . . . .	215
7.2.10	Outras Funções Úteis . . . . .	216
7.3	Fatores . . . . .	216
7.3.1	Criando Fatores . . . . .	217
7.3.2	Modificando Fatores . . . . .	218
7.3.3	Convertendo Fatores para Outras Classes . . . . .	219
7.3.4	Criando Tabelas de Contingência . . . . .	220
7.3.5	Outras Funções . . . . .	221
7.4	Valores Lógicos . . . . .	222
7.4.1	Criando Valores Lógicos . . . . .	222
7.5	Datas e Tempo . . . . .	225
7.5.1	Criando Datas Simples . . . . .	225
7.5.2	Criando Sequências de Datas . . . . .	228
7.5.3	Operações com Datas . . . . .	230
7.5.4	Lidando com Data e Tempo . . . . .	231
7.5.5	Personalizando o Formato de Datas . . . . .	233
7.5.6	Extraindo Elementos de uma Data . . . . .	235
7.5.7	Conhecendo o Horário e a Data Atual . . . . .	236
7.5.8	Outras Funções Úteis . . . . .	237
7.6	Dados Omissos - <code>NA</code> ( <i>Not available</i> ) . . . . .	239
7.6.1	Definindo Valores <code>NA</code> . . . . .	239
7.6.2	Encontrando e Substituindo Valores <code>NA</code> . . . . .	240
7.6.3	Outras Funções Úteis . . . . .	242
7.7	Exercícios . . . . .	242

<b>8</b>	<b>Programando com o R</b>	<b>247</b>
8.1	Criando Funções . . . . .	247
8.2	Utilizando Loops (comando <i>for</i> ) . . . . .	247
8.3	Execuções Condicionais ( <i>if</i> , <i>else</i> , <i>switch</i> ) . . . . .	248
8.4	Utilizando as Funções da Família <i>apply</i> . . . . .	248
8.4.1	Função <i>lapply</i> . . . . .	248
8.4.2	Função <i>sapply</i> . . . . .	248
8.4.3	Função <i>tapply</i> . . . . .	249
8.4.4	Função <i>mapply</i> . . . . .	249
8.4.5	Função <i>apply</i> . . . . .	249
8.5	Utilizando Pacote <i>purrr</i> . . . . .	249
8.5.1	Função <i>purrr::map</i> . . . . .	250
8.5.2	Função <i>purrr::safely</i> . . . . .	250
8.5.3	Função <i>purrr::pmap</i> . . . . .	250
8.6	Manipulação de Dados com <i>dplyr</i> . . . . .	250
8.6.1	Operações de Grupo com <i>dplyr</i> . . . . .	251
8.6.2	Operações de Grupo Complexas com <i>dplyr</i> . . . . .	251
8.7	Exercícios . . . . .	251
<b>9</b>	<b>Estruturando e Limpando Dados</b>	<b>253</b>
9.1	O Formato do <i>dataframe</i> . . . . .	253
9.1.1	Conversão entre <i>long</i> e <i>wide</i> . . . . .	253
9.2	Convertendo Listas em <i>dataframes</i> . . . . .	254
9.3	Eliminando Outliers . . . . .	254
9.4	Desinflationando Dados de Preços . . . . .	254
9.5	Modificando a Frequência Temporal dos Dados . . . . .	254
9.6	Exercícios . . . . .	254
<b>10</b>	<b>Visualizando Dados</b>	<b>257</b>
10.1	Criando Janelas de Gráficos com <i>x11</i> . . . . .	257
10.2	Criando Figuras com <i>qplot</i> . . . . .	257
10.3	Criando Figuras com <i>ggplot</i> . . . . .	258
10.3.1	A Curva de Juros Brasileira . . . . .	258
10.3.2	Usando Temas . . . . .	258
10.3.3	Criando Painéis com <i>facet_wrap</i> . . . . .	258
10.4	Uso do Operador <i>pipeline</i> . . . . .	259
10.5	Criando Figuras Estatísticas . . . . .	259
10.5.1	Criando Histogramas . . . . .	259
10.5.2	Criando Figuras <i>Boxplot</i> . . . . .	259
10.5.3	Criando Figuras <i>QQ</i> . . . . .	260
10.6	Salvando Figuras para Arquivos . . . . .	260
10.7	Exercícios . . . . .	260

<b>11 Econometria Financeira com o R</b>	<b>263</b>
11.1 Modelos Lineares (OLS)	263
11.1.1 Simulando um Modelo Linear	263
11.1.2 Estimando um Modelo Linear	264
11.1.3 Inferência Estatística em Modelos Lineares	264
11.2 Modelos Lineares Generalizados (GLM)	264
11.2.1 Simulando um Modelo GLM	264
11.2.2 Estimando um Modelo GLM	265
11.3 Modelos para Dados em Painel	265
11.3.1 Simulando Dados em Painel	265
11.3.2 Estimando Modelos de Dados em Painel	265
11.4 Modelos ARIMA	266
11.4.1 Simulando Modelos ARIMA	266
11.4.2 Estimando Modelos ARIMA	266
11.4.3 Prevendo Modelos ARIMA	266
11.5 Modelos GARCH	267
11.5.1 Simulando Modelos GARCH	267
11.5.2 Estimando Modelos GARCH	267
11.5.3 Prevendo Modelos GARCH	267
11.6 Modelos de Mudança de Regime	268
11.6.1 Simulando Modelos de Mudança de Regime	268
11.6.2 Estimando Modelos de Mudança de Regime	268
11.6.3 Prevendo Modelos de Mudança de Regime	268
11.7 Trabalhando com Diversos Modelos	269
11.8 Exercícios	269
<b>12 Reportando Resultados e Criando Relatórios</b>	<b>271</b>
12.1 Reportando Tabelas	271
12.2 Reportando Modelos	272
12.3 Criando Relatórios com o <i>RMarkdown</i>	272
12.4 Exercícios	272
<b>13 Otimização de Código</b>	<b>273</b>
13.1 Otimizando Código em R	273
13.2 Otimizando a Velocidade de Execução	273
13.2.1 Perfil do Código R ( <i>profiling</i> )	274
13.2.2 Estratégias para Melhorar a Velocidade de Execução	274
13.2.3 Usando Código C ++ (pacote <i>Rcpp</i> )	275
13.2.4 Usando Cacheamento Local (pacote <i>memoise</i> )	275
13.3 Exercícios	275

# Sobre Nova Edição

Desde a última edição do livro em 2018, muita coisa mudou. O ecossistema do R e RStudio tem evoluído de forma constante. Em um esforço de atualização, tenho enorme prazer em publicar a terceira edição de “Análise de Dados Financeiros e Econômicos com o R”. É gratificante perceber que, como um bom vinho, o conteúdo do livro só melhora com o tempo.

O diferencial da terceira edição é o **foco no leitor**. As principais mudanças são:

- Todo o conteúdo do livro agora é disponibilizado via **pacote adfeR**, facilitando muito a reprodução de todos os exemplos de código.
- Uso de **caixas de textos customizadas** para indicar pontos importantes e precauções que os leitores devem ter em cada seção do livro.
- Mais de **100 exercícios de final de capítulo** foram criados e agora possuem gabarito em texto e código, disponível na versão web do livro<sup>1</sup>. Todos os exercícios estão disponíveis no formato **exams** (Zeileis et al., 2022) e podem ser compilados para um pdf ou então exportados para plataformas de *e-learning*, tal como o *Moodle* ou *Blackboard* (veja seção *Conteúdo para Instrutores* no Prefácio).
- **Quatro novos pacotes** especializados na obtenção de dados financeiros e econômicos estão inclusos na nova edição. São estes: **GetDFPData2**, **GetFREData**, **GetQuandlData** e **GetBCBData**. Todos são pacotes estáveis, desenvolvidos por mim e serão mantidos ao longo do tempo. Assim, não corremos mais o risco de quebra de código devido a desatualização de um pacote por um autor.
- Um novo capítulo sobre **Otimização de Código em R**, discutindo melhorias na estrutura de código e também minimização do tempo de execução via

---

<sup>1</sup><https://www.msperlin.com/adfeR>

estratégias de cacheamento local e processamento paralelo.

- Uso de **template customizado** para o ebook e html via CSS (*Cascading Style Sheets*). Agora, o livro possui, sem dúvida, uma cara própria e consistente entre os diferentes formatos.

Este livro é um projeto vitalício que pretendo atualizar frequentemente. Vejo neste trabalho uma maneira de contribuir para a formação profissional de uma nova geração de pesquisadores e analistas de mercado. Espero que goste da obra e use-a para tornar o R um aliado no seu ciclo de trabalho.

# Prefácio

Dado que se interessou por este livro, prevejo que você é um aluno de pós-graduação dando os primeiros passos em pesquisa com dados, ou é um profissional experiente procurando conhecer novas ferramentas para utilizar em seu trabalho. Em ambos os casos, este livro é para você. **A finalidade e objeto deste trabalho é introduzir o leitor ao uso do R como ferramenta de computação e análise de dados, com uma ênfase especial para pesquisa empírica no tópico de Finanças e Economia.** Ao final deste livro você irá aprender como utilizar o R para importar e manipular dados e, por fim, reportar tabelas e figuras de uma pesquisa em um relatório técnico.

Este livro é o resultado do meu trabalho como docente na Escola de Administração da Universidade Federal do Rio Grande do Sul. No programa de pós-graduação em Administração, eu leciono uma disciplina introdutória ao uso do R para resolver problemas de pesquisa na área de Finanças e Economia. Observando os alunos em sala de aula, percebo diariamente o impacto positivo que esse tipo de orientação tem em suas futuras carreiras profissionais, seja como pesquisadores acadêmicos, seja como analistas de dados em organizações públicas e privadas. Este livro é um projeto pessoal para disseminar conhecimento sobre a ferramenta para um público maior e mais diversificado.

Outra motivação que tive para escrever o livro foi minha experiência na utilização de códigos disponibilizados por outros pesquisadores em pesquisas específicas. Na maioria das vezes, esses códigos são desorganizados, pouco claros e, possivelmente, funcionam apenas no computador do pesquisador que os escreveu! Surpreendentemente, devido a desorganização, o trabalho de desvendar o código de outros professores tende a levar mais tempo do que desenvolver eu mesmo o procedimento, por mais complexo que ele fosse. Esses casos, aliás, ferem a ciência, pois um dos princípios básicos da pesquisa é a **replicabilidade** - isto é, uma rotina de pesquisa mal escrita irá reduzir a possibilidade de outras pessoas a utilizarem.

Assim como se espera que um artigo científico esteja bem escrito, também se deve esperar que o código por trás da respectiva pesquisa seja de qualidade. Porém, esse não é o caso na grande maioria das vezes. Com este livro, irei atacar esse problema, formalizando uma estrutura de código voltada à reprodutibilidade científica, focando em organização e usabilidade. Nesse sentido, espero que as futuras gerações de pesquisadores estejam mais bem preparadas para compartilhar o seu trabalho.

Antes de prosseguir, um aviso. Não iremos trabalhar usos avançados do R. O conteúdo será limitado a exemplos simples e práticos de utilização do *software* para a construção de pesquisa baseada em dados financeiros e econômicos. De fato, um dos desafios na escrita deste livro foi definir o limite entre o material introdutório e o avançado. Procurei, sempre que possível, dosar gradualmente o nível de complexidade. Para leitores interessados em conhecer funções avançadas do programa e o seu funcionamento interno, sugiro a leitura do manual oficial do R (Teetor, 2011) e de Wickham (2019).

Com este livro irás aprender os seguinte tópicos:

**Usar o R e RStudio** O capítulo 01 apresenta, discute e justifica o uso do R como uma plataforma de programação desenhada para resolver problemas relacionados a dados. No capítulo 02 exploraremos os comandos básicos R e os recursos do RStudio, incluindo criação de objetos, execução de *scripts*, interação com o disco rígido do computador, e muito mais.

**Importação de dados financeiros e econômicos** Nos capítulos 04 e 05 vamos aprender a importar dados de arquivos locais, tal como uma planilha do Excel, ou então da internet. Aprenderemos a tirar proveito da modularidade do R e, quando necessário, instalar pacotes que permitam o download de dados diversos tal como preços de ações, índices econômicos, curva de juros, dados financeiros de empresas e muito mais.

**Limpar, estruturar e analisar dados** Nos capítulos 06 e 07 iremos estudar o ecossistema de objetos do R. Aprenderemos a manipular objetos tal como vetores numéricos, datas e tabelas inteiras. Nos capítulos 08 e 09 vamos estudar o uso de ferramentas de programação para resolver problemas com dados incluindo limpeza, reestruturação e também análise.

**Visualização de dados** No capítulo 10 aprenderemos a usar o pacote `ggplot2` para criar visualizações do conjunto de dados, incluindo os casos mais comuns em finanças e economia, séries temporais e gráficos estatísticos.

**Analisar dados com econometria** No capítulo 11 aprenderemos a usar os modelos econométricos mais populares em finanças e economia, incluindo o modelo linear, GLM, Arima e outros. Isto inclui simulação, teste de hipóteses e estimação de modelos para diversas séries de dados reais.

**Reporte de resultados** No capítulo 12 veremos como reportar os resultados de sua pesquisa para um documento externo com a exportação fácil e reproduzível de tabelas e figuras. O conteúdo também inclui uma seção sobre a inovadora

tecnologia *RMarkdown*, a qual permite que código e texto sejam compilados conjuntamente.

**Melhorando o seu código** No último capítulo do livro vamos discutir as melhores práticas de programação, incluindo análise do perfil de execução do código R, destacando pontos de gargalo e melhoria do tempo de execução com estratégias de cacheamento local, uso de código C++ e processamento paralelo.

## Material Suplementar

Todo o material usado no livro, incluindo exemplos de código separados por capítulos, está publicamente disponível na Internet e distribuído com um pacote R denominado `adfeR`. Este inclui arquivos de dados, código em si, e várias funções que irão facilitar a execução dos exemplos do livro. Se você planeja, como sugerido, escrever código enquanto lê o livro, este pacote ajudará muito em sua jornada.

Para instalar este pacote no seu computador, basta executar algumas linhas de comando no R. Veja o código destacado a seguir e copie e cole o mesmo no prompt do RStudio (canto inferior esquerdo da tela, com um sinal “>”) e pressione Enter para cada comando. Esteja ciente de que você precisará do R e RStudio instalados em seu computador (consulte a seção 1.4).

```
# install devtools dependency
install.packages('devtools')

# install book package
devtools::install_github('msperlin/adfeR')
```

O que este código fará é instalar o pacote `devtools`, uma dependência necessária para instalar código do Github – um repositório de pacotes onde o livro está hospedado. Depois disso, uma chamada para `install_github('msperlin/adfeR')` irá instalar o pacote em seu computador.

Depois da instalação, todos os arquivos do livro estarão disponíveis localmente, salvos em uma pasta do seu computador. Iremos usar todos estes arquivos futuramente. Opcionalmente, caso quiser olhar os arquivos, podes copiar todo conteúdo para outra pasta com o código a seguir:

```
adfeR::copy_book_files(path_to_copy = '~')
```

Veja que o tilda (~) é um atalho para o diretório “Documentos” no Windows (ou “home” no Linux/Mac). Assim, o código anterior descompactará o arquivo do livro na pasta “Documentos/adfeR-files”. O pacote também inclui várias outras funções que serão usadas ao longo do livro. Se você preferir a maneira antiga e consagrada de baixar o arquivo e descompactar manualmente, podes encontrar uma cópia no

site do livro<sup>2</sup>.

## Conteúdo para Instrutores

Se você for um instrutor de R, aqui encontrará muito material para usar em suas aulas:

**Exercícios estáticos na internet** Cada capítulo deste livro inclui exercícios que seus alunos podem praticar. Todas as soluções estão disponíveis na versão online do livro, disponível em <https://www.msperlin.com/adfeR/>.

**Exercícios exportáveis para pdf ou plataformas de *e-learning*** Todos os exercícios do livro estão no formato `exams` (Zeileis et al., 2022) e são exportáveis para arquivos em pdf ou então para plataformas de *e-learning* tal como o *Moodle* ou *Blackboard*. Veja este post no blog<sup>3</sup> para maiores detalhes.

**Acesso ao livro na internet** Na versão web do livro, o conteúdo integral está liberado até o capítulo 7, o qual é mais que suficiente para um curso introdutório sobre a plataforma.

Espero que goste deste livro. O conteúdo tem sido compilado por um longo período de tempo, a base de muito suor e, literalmente, litros de café por parte do autor.

Boa leitura!

Marcelo S. Perlin

---

<sup>2</sup><https://www.msperlin.com/files/adfer-files/adfeR-code-and-data.zip>

<sup>3</sup><https://www.msperlin.com/post/2021-02-18-dynamic-exercises-adfer/>

# Capítulo 1

## Introdução

**Na era digital, informação é abundante e de fácil acesso.** Nota-se, em todas as áreas de conhecimento, um crescimento substancial no registro digital dos mais diversos eventos. A cada segundo, volumosos bancos de dados de diferentes empresas e organizações são alimentados com novas informações. Essa tendência impõe uma alteração significativa na forma como organizações utilizam a análise de dados na tomada de decisões. Sem dúvida, o período atual é bastante prolífico para profissionais com conhecimento e experiência na utilização das ferramentas corretas para a análise computacional de dados.

Em particular, a área de Economia e Finanças oferece grande variedade de informações disponíveis ao público. Instituições de pesquisa como IBGE, IPEA, Bancos Centrais, bolsas de valores e tantas outras, disponibilizam seus dados publicamente, seja por obrigatoriedade legal, seja para o próprio fomento da atividade de pesquisa. Hoje em dia, computadores domésticos já possuem a capacidade de processar quantidades volumosas de dados em pouco tempo. Essa evolução do aparato computacional ocorreu mais rapidamente do que o acréscimo de complexidade das metodologias utilizadas, resultando em uma diminuição significativa do tempo necessário para a execução de uma pesquisa. Ou seja, o volume de dados não é mais uma barreira de entrada para analistas.

Os métodos de processamento também avançaram em termos de complexidade. No passado, uma simples planilha eletrônica poderia dar conta do trabalho de análise. Hoje em dia, é esperado que um analista de dados ou aluno de pós-graduação saiba programar e analisar dados via código, facilitando e acelerando a obtenção de resultados, permitindo a colaboração via reproducibilidade de resultados.

É nesse ambiente que se destaca o papel do R, uma linguagem de programação voltada para a resolução de problemas computacionais envolvendo análise, processa-

mento, visualização e modelagem de dados. Nas próximas seções, explicarei o que é o R e quais são suas vantagens frente a outras alternativas.

## 1.1 O que é o R

O R é uma linguagem de programação voltada para a resolução de problemas estatísticos e para a visualização gráfica de dados. O código base do R foi inspirado na linguagem *S*, inicialmente criada no laboratório da **Bell/AT&T** por **John Chambers** e seus colegas. Esse código foi reaproveitado por dois acadêmicos, **Ross Ihaka** e **Robert Gentleman**, resultando na plataforma de programação que temos hoje. Para os curiosos, o nome *R* foi escolhido devido ao compartilhamento da primeira letra do nome de seus criadores.

Hoje, R é sinônimo de programação voltada à análise de dados, com uma larga base de usuários e funções bem estabelecidas. É muito provável que pesquisadores de áreas diversas, desde Economia até Biologia, ou mesmo Música, encontrem no R uma quantidade significativa de códigos que facilitem suas análises. No campo empresarial, grandes empresas como *Google* e *Microsoft* já o adotaram como a linguagem interna para a análise de dados. O R é atualmente mantido pelo **R Foundation** e o **R Consortium**, um esforço coletivo para financiar projetos de extensão da linguagem.

E o mais importante: **o R é totalmente livre** e disponível em vários sistemas operacionais. Seja você um usuário do Windows, do Linux/Unix ou do MacOS, existe uma instalação do R para a sua plataforma, e os seus códigos devem rodar entre uma e outra com mínimas alterações.

## 1.2 Por que Escolher o R

O processo de aprendizado de uma nova linguagem de programação exige muito tempo e esforço. Portanto, é importante entender as razões por trás dessa escolha. Possivelmente você esteja se perguntando por que deve optar pelo R e investir tempo em sua aprendizagem, ao invés de escolher uma outra linguagem.

Em primeiro lugar, **o R é uma plataforma madura, estável, continuamente suportada e intensamente utilizada na indústria.** Ao escolher o R, você terá a bagagem computacional necessária não somente para uma carreira acadêmica em pesquisa científica, mas também para o trabalho em organizações privadas. Nesse sentido, com a escolha de outra linguagem de programação menos popular ou proprietária/comercial, é provável que tal linguagem não seja utilizada em um ambiente empresarial e isso pode limitar as suas oportunidades profissionais futuras. Uma bagagem de conhecimento em R certamente aumenta a sua atratividade como um profissional em Finanças e Economia.

**Aprender a linguagem do R é fácil.** A experiência que tenho ensinando o R em sala de aula me permite afirmar que os alunos, mesmo aqueles sem experiência em programação, apresentam facilidade em aprender a linguagem e em utilizá-la para criar seus próprios códigos de pesquisa. A linguagem é intuitiva e certas normas e funções podem ser estendidas para diferentes casos. Após entender como o programa funciona, fica fácil descobrir novas funcionalidades partindo de uma lógica anterior. Essa notação compartilhada entre procedimentos facilita o aprendizado.

**A interface do R e RStudio torna o uso da ferramenta bastante produtivo.** A interface gráfica disponibilizada pelo RStudio facilita o uso do software, assim como a produtividade do usuário. Utilizando o ambiente de trabalho do R e RStudio, têm-se a disposição diversas ferramentas que facilitam e estendem o uso da plataforma.

**Os pacotes do R permitem as mais diversas funcionalidades.** Logo veremos que o R permite o uso de código de outros usuários, os quais podem ser localmente instalados através de um simples comando. Esses estendem a linguagem básica do R e possibilitam as mais diversas funcionalidades. Podes, por exemplo, utilizar o R para mandar emails estruturados, escrever e publicar um livro, criar provas objetivas com conteúdo dinâmico, contar piadas e poemas (é sério!), acessar e coletar dados da internet, entre diversas outras funcionalidades.

**O R tem compatibilidade com diferentes linguagens e sistemas operacionais.** Se, por algum motivo, o usuário precisar utilizar código de outra linguagem de programação tal como *C++*, *Python*, *Julia*, é fácil integrar a mesma dentro de um programa do R. Diversos pacotes estão disponíveis para facilitar esse processo. Portanto, o usuário nunca fica restrito a uma única linguagem e tem flexibilidade para escolher as suas ferramentas de trabalho.

**O R é totalmente gratuito!** O programa e todos os seus pacotes são completamente livres, não tendo custo algum de licença e distribuição. Portanto, você pode utilizá-lo e modificá-lo livremente no seu trabalho ou computador pessoal. Essa é uma razão muito forte para a adoção da linguagem em um ambiente empresarial, onde a obtenção de licenças individuais e coletivas de outros softwares comerciais pode incidir um alto custo financeiro.

### 1.3 Usos do R

O R é uma linguagem de programação completa e qualquer problema computacional pode ser resolvido com base nela. Dada a adoção do R por diferentes áreas de conhecimento, a lista de possibilidades é extensa. Para o caso de Finanças e Economia, destaco abaixo possíveis utilizações do programa:

- Substituir e melhorar tarefas intensivas e repetitivas dentro de ambientes cor-

porativos, geralmente realizadas em planilhas eletrônicas;

- Desenvolvimento de rotinas para administrar portfólios de investimentos e executar ordens financeiras;
- Criação de ferramentas para controle, avaliação e divulgação de índices econômicos sobre um país ou região;
- Execução de diversas possibilidades de pesquisa científica através da estimação de modelos econométricos e testes de hipóteses;
- Criação e manutenção de *websites* dinâmicos ou estáticos através dos pacotes `shiny`, `blogdown` ou `distill`;
- Organização de um processo automatizado de criação e divulgação de relatórios técnicos com o pacote `knitr` e a tecnologia *RMarkdown*.

Além dos usos destacados anteriormente, o acesso público a pacotes desenvolvidos por usuários expande ainda mais essas funcionalidades. O site da CRAN (*Comprehensive R Archive Network*)<sup>1</sup> oferece um *Task Views* do software para o tópico de Finanças<sup>2</sup> e Econometria<sup>3</sup>. Nos links é possível encontrar os principais pacotes disponíveis para cada tema. Isso inclui a importação de dados financeiros da internet, a estimação de um modelo econométrico específico, cálculos de diferentes estimativas de risco, entre várias outras possibilidades. A leitura dessa página e o conhecimento desses pacotes são essenciais para aqueles que pretendem trabalhar com Finanças e Economia. Vale destacar, porém, que essa lista é moderada e apresenta apenas os principais itens. A lista completa de pacotes é muito maior do que o apresentado no *Task Views*.

## 1.4 Como Instalar o R

O R é instalado no seu sistema operacional como qualquer outro programa. A maneira mais direta e funcional de instalá-lo é ir ao website do R em <https://www.r-project.org/> e clicar no *link CRAN* do painel *Download*, conforme mostrado na figura a seguir.

A próxima tela apresenta a escolha do espelho para baixar os arquivos de instalação. O repositório do CRAN é espelhado em diversas partes do mundo, permitindo acesso rápido para os usuários. Para a grande maioria dos leitores deste livro, essa localidade deve ser o Brasil. Portanto, você pode escolher um dos links da instituição mais próxima, tal como o da UFPR (Universidade Federal do Paraná). Em

---

<sup>1</sup><https://cran.r-project.org/web/views>

<sup>2</sup><https://cran.r-project.org/web/views/Finance.html>

<sup>3</sup><https://cran.r-project.org/web/views/Econometrics.html>

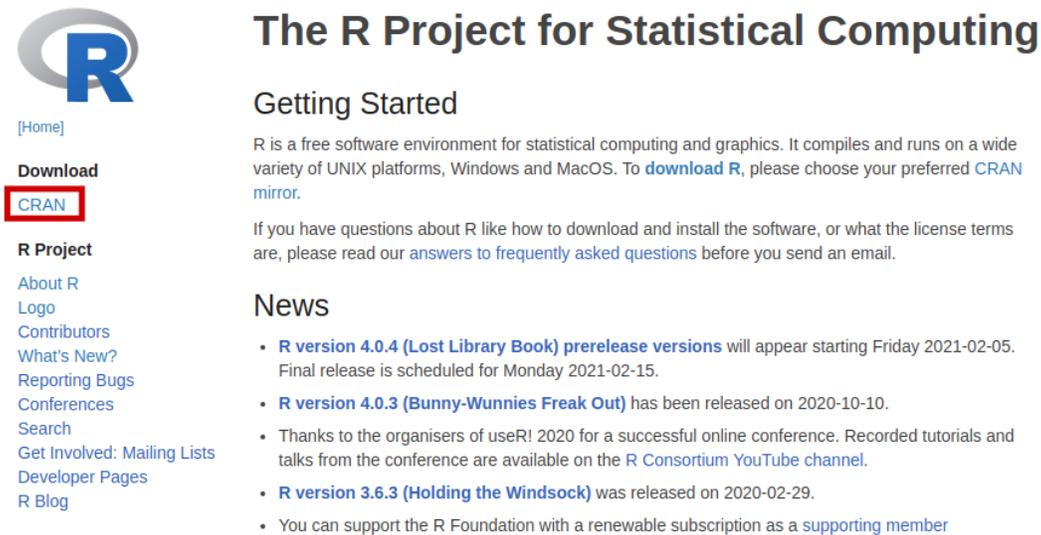


Figura 1.1: Página inicial para o download do R

caso de dúvida, escolha o repositório do RStudio 0-Cloud (veja Figura 1.2), o qual automaticamente direciona para o local mais próximo.

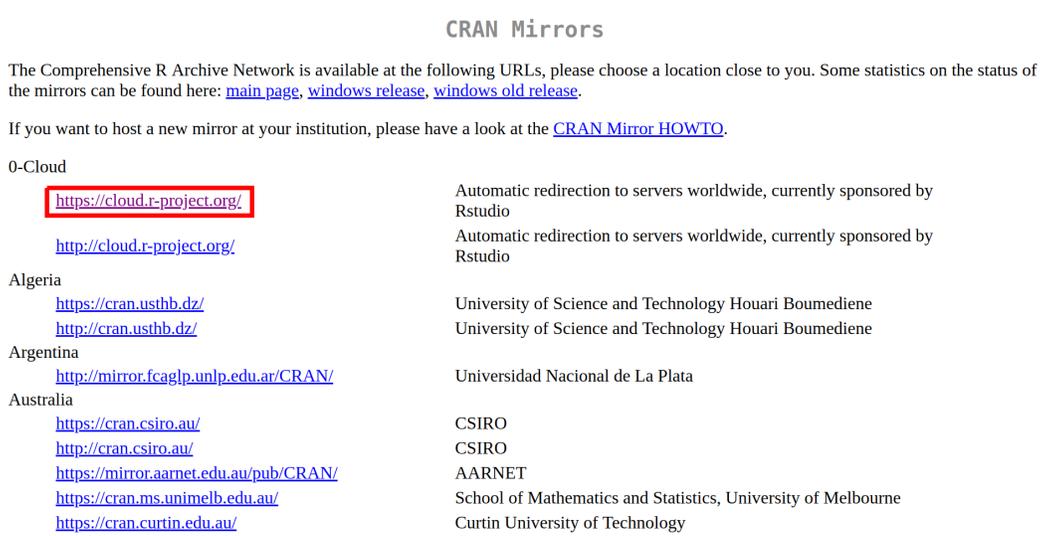
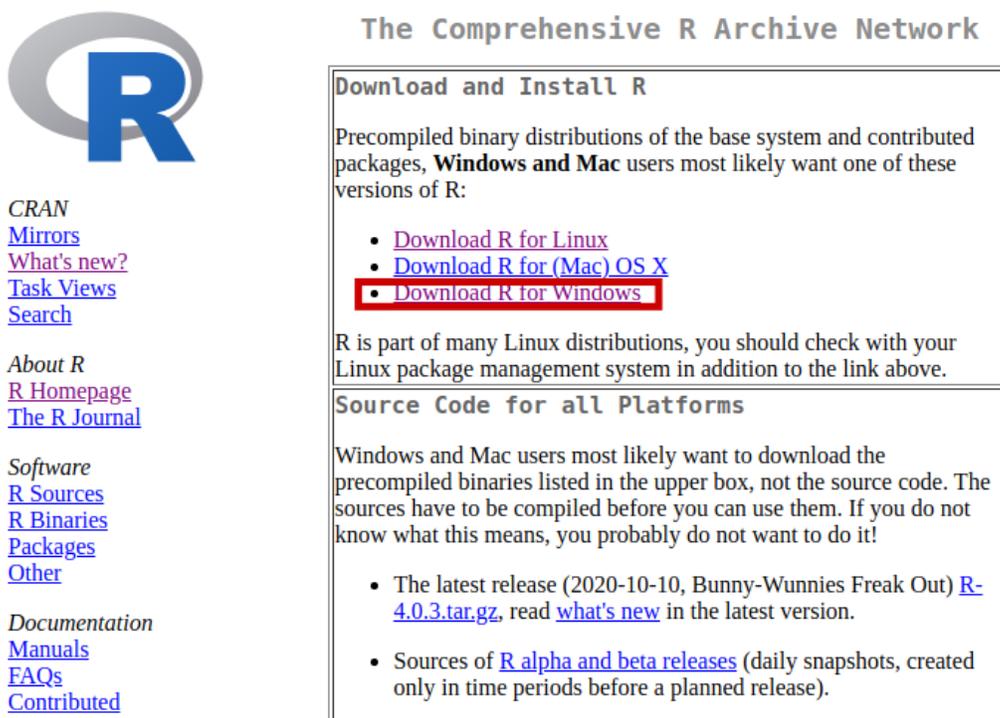


Figura 1.2: Tela com a escolha do espelho para o download

O próximo passo é selecionar o sistema operacional do computador. Devido à maior popularidade da plataforma *Windows*, a partir de agora daremos enfoque à instalação do R nesse sistema. As instruções de instalação nos demais sistemas operacio-

nais podem ser facilmente encontradas na internet. Destaca-se que, independente da plataforma, o modo de uso do R é o mesmo. Existem, porém, algumas exceções, principalmente quando o R interage com o sistema de arquivos. Essas exceções serão destacadas no decorrer do livro. Assim, mesmo que você esteja utilizando Linux ou MacOS, poderá tirar proveito do material aqui apresentado.



**The Comprehensive R Archive Network**

**Download and Install R**

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

**Source Code for all Platforms**

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2020-10-10, Bunny-Wunnies Freak Out) [R-4.0.3.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).

Figura 1.3: Tela com a escolha do sistema operacional

Após clicar no link *Download R for Windows*, a próxima tela irá mostrar as seguintes opções de *download*: *base*, *contrib*, *old.contrib* e *RTools*. Dentre as opções de *download*, a primeira (*base*) deve ser selecionada. O *link* acessa a instalação básica do R para *Windows*. O link *contrib* e *old.contrib* acessa os pacotes/módulos disponíveis para o R. Não precisas acessar estes últimos links, existe uma maneira muito mais fácil de instalar pacotes, como veremos em seguida.

O último link, *RTools*, serve para instalar dependências necessárias no caso do usuário desenvolver e distribuir os seus próprios pacotes de R. Este não é uma instalação necessária para usuários iniciantes. Porém, saiba que alguns pacotes externos ao CRAN podem exigir a instalação do *RTools* para compilação de código. Minha sugestão é que já instale este software e assim evite qualquer problema futuro.

Após clicar no link *base*, a próxima tela mostrará o link para o *download* do arquivo de instalação do R no *Windows*.



CRAN  
[Mirrors](#)  
[What's new?](#)  
[Task Views](#)  
[Search](#)

About R  
[R Homepage](#)  
[The R Journal](#)

Software  
[R Sources](#)  
[R Binaries](#)  
[Packages](#)  
[Other](#)

Documentation  
[Manuals](#)  
[FAQs](#)  
[Contributed](#)

## R for Windows

Subdirectories:

<a href="#">base</a>	Binaries for base distribution. This is what you want to <a href="#">install R for the first time</a> .
<a href="#">contrib</a>	Binaries of contributed CRAN packages (for R >= 2.13.x; managed by Uwe Ligges). There is also information on <a href="#">third party software</a> available for CRAN Windows services and corresponding environment and make variables.
<a href="#">old contrib</a>	Binaries of contributed CRAN packages for outdated versions of R (for R < 2.13.x; managed by Uwe Ligges).
<a href="#">Rtools</a>	Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself.

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

Note: CRAN does some checks on these binaries for viruses, but cannot give guarantees. Use the normal precautions with downloaded executables.

Figura 1.4: Tela com opções de instalação



CRAN  
[Mirrors](#)  
[What's new?](#)  
[Task Views](#)  
[Search](#)

About R  
[R Homepage](#)  
[The R Journal](#)

Software  
[R Sources](#)  
[R Binaries](#)  
[Packages](#)  
[Other](#)

## R-4.0.3 for Windows (32/64 bit)

[Download R 4.0.3 for Windows](#) (85 megabytes, 32/64 bit)  
[Installation and other instructions](#)  
[New features in this version](#)

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the [md5sum](#) of the .exe to the [fingerprint](#) on the master server. You will need a version of md5sum for windows: both [graphical](#) and [command line versions](#) are available.

### Frequently asked questions

- [Does R run under my version of Windows?](#)
- [How do I update packages in my previous version of R?](#)
- [Should I run 32-bit or 64-bit R?](#)

Please see the [R FAQ](#) for general information about R and the [R Windows FAQ](#) for Windows-specific information.

### Other builds

Figura 1.5: Tela para o download do R

Após baixar o arquivo, abra-o e siga os passos da tela de instalação do R. Escolha a língua inglesa em todas etapas do processo. O uso da língua inglesa não é acidental. Este é a melhor forma, mesmo para iniciantes, de se aprender a usar o R. É possível instalar uma versão em português porém isso limita o potencial da ferramenta. Caso não for fluente em inglês, não se preocupe, o vocabulário necessário é básico. Neste momento, nenhuma outra configuração especial é necessária. Sugiro manter todas as escolhas padrão selecionadas e simplesmente ir aceitando as telas de diálogo. Após a instalação do R, partimos para a instalação do RStudio.



A cada quatro meses uma nova versão do R é lançada, corrigindo *bugs* e implementando novas soluções. Temos dois tipos principais de versões, *major* e *minor*. Por exemplo, hoje, 05/02/2021, a última versão do R é 4.0.3. O primeiro dígito (“4”) indica a versão *major* e todos os demais são do tipo *minor*. Geralmente, as mudanças *minor* são bem específicas e, possivelmente, terão pouco impacto no seu trabalho.

Porém, ao contrário de mudanças *minor*, mudanças do tipo *major* refletem totalmente no ecossistema de pacotes do R. Toda vez que instalar uma nova versão *major* do R, terá que reinstalar todos os pacotes utilizados. O problema é que não é incomum problemas de incompatibilidade de pacotes com a nova versão.

Minha dica é: toda vez que uma nova versão *major* do R sair, espere alguns meses antes de instalar na sua máquina. Assim, o autores dos pacotes terão mais tempo para atualizar os seus códigos, minimizando a possibilidade de problemas de compatibilidade.

## 1.5 Instalando o RStudio

A instalação do R inclui a sua própria interface gráfica, um programa que facilita a edição e execução de nossos *scripts*. Essa, porém, possui várias limitações. O RStudio é um *software* que torna o uso e o visual do R muito mais prático e eficiente. Uma forma de entender essa relação é com uma analogia com carros. Enquanto o R é o motor da linguagem de programação, o RStudio é a carroceria e o painel de instrumentos, os quais melhoram significativamente a experiência de uso. Além de apresentar um visual mais atrativo, o RStudio também é acrescido de várias funcionalidades que facilitam a vida do usuário, possibilitando a construção de projetos e pacotes do próprio R, a criação de documentos dinâmicos (*Sweave/knitr*) e a interface com edição de textos em *LaTeX*, entre várias outras. Assim como o R, **o RStudio também é gratuito** e pode ser utilizado no ambiente empresarial.

A instalação do RStudio é mais simples do que a do R. Os arquivos estão disponíveis no endereço disponibilizado no site do livro. Após acessar a página, clique em *Download RStudio* e depois em *Download Rstudio Desktop*. Logo após, basta

selecionar o arquivo relativo ao sistema operacional em que você irá trabalhar. Provavelmente, essa opção será *Windows Vista/7/8/10*. Note que, assim como o R, o RStudio também está disponível para diferentes plataformas.

Destaco que o uso do RStudio não é essencial para desenvolver programas no R. Outros softwares de interface estão disponíveis e podem ser utilizados. Porém, dada minha experiência atual, o RStudio é o programa de interface que oferece a maior variedade de funcionalidades para essa linguagem, além de ser amplamente utilizado, o que justifica a sua escolha.

## 1.6 Recursos na Internet

**A comunidade R é viva e envolvente.** Na internet é possível encontrar uma diversidade de material sobre o uso do R. Diversos usuários, assim como o próprio autor do livro, publicam material sobre o uso R em seus blogs. Isso inclui anúncios de pacotes, publicações sobre análise de dados na vida real, curiosidades, novidades e tutoriais. **R-Bloggers** é um site internacional que agrega esses blogs em um único local, tornando mais fácil para qualquer um acessar e participar. O conteúdo do R-Bloggers, porém, é todo em inglês.

Recentemente, uma lista de blogs locais sobre o R está compilada e organizada por Marcos Vital em <https://marcosvital.github.io/blogs-de-R-no-Brasil/>. Eu recomendo a inscrição no feed do R-Bloggers, além dos blogs nacionais. Não só você será informado sobre o que está acontecendo no universo do R, mas também aprenderá muito lendo artigos e os códigos de outros usuários.

Aprender e usar R pode ser uma experiência social. Várias conferências e grupos de usuários estão disponíveis em muitos países, incluindo o Brasil. O grupo *R Brasil - Programadores* no Facebook é bastante ativo, com um grande número de participantes. Recomendo fortemente a inscrição neste grupo e o acompanhamento das discussões relacionadas ao uso do R. Diversas conferências locais sobre o R são divulgadas nesse grupo.

## 1.7 Organização e Material do Livro

Este livro tem uma abordagem prática no uso do R e será acompanhado por uma série de códigos que irão exemplificar e mostrar para o leitor as funcionalidades do programa. Para tirar o máximo de proveito do material, sugiro que você primeiro busque entender o código mostrado e, somente então, tente utilizá-lo em seu próprio computador. O índice remissivo disponibilizado no final do livro serve como uma mapa de uso das funções. Toda vez que uma função é chamada no livro, um registro do número da página é criado no índice remissivo. Esse indica, também, o pacote que a função pertence. Podes utilizar este mapa para localizar o uso de qualquer

função ou pacote no decorrer do livro.

Sugiro também o uso da versão web do livro<sup>4</sup>, a qual permite que os códigos de exemplo sejam copiados direto para a sua sessão do R. Assim, perderás menos tempo digitando código, sobrando tempo para o estudo.

Aprender a programar em uma nova linguagem é como aprender uma língua estrangeira: o uso no dia-a-dia é de extrema importância para criar fluência. Sempre que possível, teste o código no seu computador e *brinque* com o mesmo, modificando os exemplos dados e verificando o efeito das modificações nas saídas do programa. Procure sempre entender como a rotina estudada pode ajudar na solução de um problema seu. Cada capítulo apresenta no seu final uma lista de exercícios, incluindo questões do tipo desafio. Podes testar as suas habilidades de programação resolvendo as atividades propostas. Vale lembrar que todo o código deste livro está disponibilizado na internet. Não precisas, portanto, escrever o código diretamente do livro. Podes copiar e colar do código fonte disponibilizado no site.

No decorrer da obra, toda demonstração de código terá duas partes: o código em si e sua saída do R. Essa saída nada mais é do que o resultado dos comandos na tela do programa. Todas as entradas e saídas de código serão sinalizadas no texto com um formato especial. Veja o exemplo a seguir:

```
# create a list
L <- list(var1 = 'abc', var2 = 1:5)

# print to prompt
print(L)
```

```
R> $var1
R> [1] "abc"
R>
R> $var2
R> [1] 1 2 3 4 5
```

No caso anterior, os textos `L <- list(var1 = 'abc', var2 = 1:5)` e `print(L)` são os códigos de entrada. A saída do programa é a apresentação na tela dos elementos de `x`, com o símbolo antecessor `R>`. Por enquanto não se preocupe em entender e reproduzir o código utilizado acima. Iremos tratar disso no próximo capítulo.

Note que faço uso da língua inglesa no código, tanto para a nomeação de objetos quanto para os comentários. Isso não é acidental. O uso da língua inglesa facilita o desenvolvimento de código ao evitar caracteres latinos, além de ser uma das línguas mais utilizadas no mundo. Portanto, é importante já ir se acostumando com esse

---

<sup>4</sup><https://www.msperlin.com/adfeR>

formato. O vocabulário necessário, porém, é limitado. De forma alguma precisarás ter fluência em inglês para entender o código.

O código também pode ser espacialmente organizado usando novas linhas. Esse é um procedimento comum em torno de argumentos de funções. O próximo pedaço de código é equivalente ao anterior, e executará exatamente da mesma maneira. Observe como usei uma nova linha para alinhar verticalmente os argumentos da função `list`. Você verá em breve que, ao longo do livro, esse tipo de alinhamento vertical é constantemente usado em códigos longos. Afinal, o código tem que necessariamente caber na página do livro.

```
# create a list
L <- list(var1 = 'abc',
          var2 = 1:5)

# print to prompt
print(L)
```

```
R> $var1
R> [1] "abc"
R>
R> $var2
R> [1] 1 2 3 4 5
```

O código também segue uma estrutura bem definida. Uma das decisões a ser feita na escrita de códigos de computação é a forma de nomear os objetos e como lidar com a estrutura do texto do código em geral. É recomendável seguir um padrão limpo de código, de forma que o mesmo seja fácil de ser mantido ao longo do tempo e de ser entendido por outros usuários. Para este livro, foi utilizado uma mistura de escolhas pessoais do autor com o estilo de código sugerido pelo Google. O usuário, porém, é livre para escolher a estrutura que achar mais eficiente. Voltaremos a discutir estrutura de código no capítulo 13.

## 1.8 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - A linguagem R foi desenvolvida com base em qual outra linguagem de programação?

- a) Python
- b) Javascript
- c) S
- d) C++
- e) Julia

02 - Qual o nome dos dois autores do R?

- a) Guido van Rossum e Bjarne Stroustrup
- b) Roger Federer e Rafael Nadal
- c) John Chambers e Robert Engle
- d) Linus Torvalds e Richard Stallman
- e) Ross Ihaka e Robert Gentleman

03 - Qual o principal diferencial do R em relação a outras linguagens de programação, tal como Python, C++, javascript e demais?

- a) Execução rápida de códigos
- b) Desenvolvimento mobile
- c) Facilita o desenvolvimento de aplicativos web
- d) Facilita a análise de dados
- e) Facilidade de uso geral

04 - Qual a razão para o nome da linguagem de programação ser R?

- a) Uma letra foi sorteada e o resultado foi R.
- b) A mãe de um dos autores se chamada Renata e, por isso, ele homenageou-a com o nome R.
- c) Era a única letra ainda não usada como linguagem de programação.
- d) R = Reusable code.
- e) Compartilhamento da letra R por seus autores.

05 - Sobre o R, considere as seguintes alternativas:

I - O R é uma plataforma de programação madura e estável;

II - O RStudio é uma interface ao R, aumentando a produtividade do analista;

III - O R tem compatibilidade com diferentes linguagens de programação;

Quais alternativas estão corretas?

- a) TRUE, FALSE, FALSE
- b) FALSE, FALSE, FALSE
- c) FALSE, TRUE, TRUE
- d) FALSE, TRUE, FALSE
- e) TRUE, TRUE, TRUE

06 - Assim que tiver R e RStudio instalado, dirija-se ao site de pacotes do CRAN<sup>5</sup> e procure por tecnologias que usas no seu trabalho. Por exemplo, se usas Planilhas do Google (Sheets)<sup>6</sup> ostensivamente no seu trabalho, logo descobrirá que existe um pacote no CRAN que interage com planilhas na nuvem.

<sup>5</sup>[https://cloud.r-project.org/web/packages/available\\_packages\\_by\\_date.html](https://cloud.r-project.org/web/packages/available_packages_by_date.html)

<sup>6</sup><https://www.google.com/sheets/about/>

07 - No site de instalação do R no Windows é possível instalar também o aplicativo Rtools. Para que ele serve?

- a) Fazer café (?).
- b) Criar gráficos.
- c) Compilar relatórios técnicos!
- d) Compilação de pacotes.
- e) Construir páginas na web.

08 - Use o Google para pesquisar por grupos de R em sua região. Verifique se os encontros são frequentes e, caso não tiver um impedimento maior, vá para um desses encontros e faça novos amigos.

09 - Dirija-se ao site do RBloggers<sup>7</sup> e procure por um tópico do seu interesse, tal como futebol (*soccer*) ou investimentos (*investments*). Leia pelo menos três artigos encontrados.

10 - Caso trabalhe em uma instituição com infraestrutura de dados, converse com o encarregado do IT e busque entender quais são as tecnologias empregadas. Verifique se, através do R, é possível ter acesso a todas tabelas dos bancos de dados. Por enquanto não existe necessidade de escrever código, ainda. Apenas verifique se esta possibilidade existe.

---

<sup>7</sup><https://www.r-bloggers.com/>



# Capítulo 2

## Operações Básicas no R

Antes de começar a desenvolver o seu código, é necessário entender a forma de trabalhar com o R e o RStudio. Isso inclui os componentes da linguagem, os diferentes tipos de objetos e as operações que podem ser realizadas com base nos comandos existentes.

Neste capítulo iremos percorrer os passos iniciais sobre o ponto de vista de alguém que nunca trabalhou com o R e, possivelmente, nunca teve contato com outra linguagem de programação. Aqueles já familiarizados com o programa irão encontrar pouca informação nova e, portanto, sugiro a leitura da próxima seção. É recomendado, porém, que no mínimo esse usuário verifique os temas tratados para confirmar ou não o seu conhecimento sobre as funcionalidades do programa. Muitas das sugestões apresentadas aqui tem potencial de aumentar significativamente a sua produtividade no RStudio.

### 2.1 Como o R Funciona?

A maior dificuldade que um usuário iniciante possui ao começar a desenvolver rotinas com o R é a forma de trabalho. A nossa interação com computadores foi simplificada ao longo dos anos e atualmente estamos confortáveis com o formato de interação do tipo *aponte e clique*. Isto é, caso se queira efetuar alguma operação no computador, basta apontar o *mouse* para o local específico na tela e clicar um botão que realize tal operação. Uma série de passos nesse sentido permite a execução de tarefas complexas no computador. Mas não se engane, essa forma de interação no formato *aponte e clique* é apenas uma camada por cima do que realmente acontece no computador. Por trás de todo *clique* existe um comando sendo executado, seja na abertura de um arquivo *pdf*, direcionamento do *browser* para uma página na internet ou qualquer outra operação cotidiana de trabalho.

Enquanto esse formato de interação visual e motora tem seus benefícios ao facilitar e popularizar o uso de computadores, é pouco flexível e eficaz quando se trabalha com procedimentos computacionais. Ao conhecer os possíveis comandos disponíveis ao usuário, é possível criar um arquivo contendo alguns comandos em sequência e, futuramente, simplesmente pedir que o computador **execute** esse arquivo com os nossos procedimentos. **Uma rotina de computador é nada mais do que um texto que instrui, de forma clara e sequencial, o que o computador deve fazer.** Investe-se certo tempo para a criação do programa, porém, no futuro, esse irá executar sempre da mesma maneira o procedimento gravado. No médio e longo prazo, existe um ganho significativo de tempo entre o uso de uma rotina do computador e uma interface do tipo *aponte e clique*.

Além disso, o **risco de erro humano na execução do procedimento é quase nulo**, pois os comandos e a sua sequência estão registrados no arquivo texto e irão ser executados sempre da mesma maneira. Da mesma forma, esse aglomerado de comandos pode ser compartilhado com outras pessoas, as quais podem replicar os resultados em seus computadores. Essa é uma das grandes razões que justificam a popularização de programação na realização de pesquisa em dados. Todos os procedimentos executados podem ser replicados pelo uso de um *script*.

No uso do R, o ideal é mesclar o uso do mouse com a utilização de comandos. O R e o RStudio possuem algumas funcionalidades através do *mouse*, porém a sua capacidade é otimizada quando os utilizamos via inserção de comandos específicos. Quando um grupo de comandos é realizado de uma maneira inteligente, temos um *script* do R que deve preferencialmente produzir algo importante para nós no final de sua execução. Em Finanças e Economia, isso pode ser o valor atualizado de um portfólio de investimento, o cálculo de um índice de atividade econômica, a performance histórica de uma estratégia de investimento, o resultado de uma pesquisa acadêmica, entre diversas outras possibilidades.

O R também possibilita a exportação de arquivos, tal como figuras a serem inseridas em um relatório técnico ou informações em um arquivo texto. De fato, o próprio relatório técnico pode ser dinamicamente criado dentro do R através da tecnologia *RMarkdown*. Por exemplo, este livro que estás lendo foi escrito utilizando o pacote `bookdown` (Xie, 2022), o qual é baseado em *RMarkdown*. O conteúdo do livro é compilado com a execução dos códigos e as suas saídas são registradas em texto. Todas as figuras e os dados do livro podem ser atualizados com a execução de um simples comando.

O produto final de trabalhar com R e RStudio será um script que produz elementos para um relatório de dados. Um bom exemplo de um código simples e polido pode ser encontrado neste link<sup>1</sup>. Abra-o e você verá o conteúdo de um arquivo com extensão *.R* que fará o download dos preços das ações de duas empresas e criará

---

<sup>1</sup>[https://github.com/msperlin/adfeR/raw/master/inst/extdata/others/S\\_Example\\_Script.R](https://github.com/msperlin/adfeR/raw/master/inst/extdata/others/S_Example_Script.R)

um gráfico e uma tabela. Ao terminar de ler o livro, você irá entender o que está acontecendo no código e como ele realiza o trabalho. Melhor ainda, você poderá melhorá-lo com novas funcionalidades e novas saídas. Caso esteja curioso em ver o script rodar, faça o seguinte: 1) instale R e RStudio no computador, 2) copie o conteúdo de texto do link para um novo script (“File” -> “New File” -> “R Script”), 3) salve-o com um nome qualquer e, finalizando, 4) pressione `control + shift + enter` para executar o *script* inteiro.

## 2.2 Objetos e Funções

No R, **tudo é um objeto, e cada tipo de objeto tem suas propriedades**. Por exemplo, o valor de um índice de inflação ao longo do tempo – em vários meses e anos – pode ser representado como um objeto do tipo vetor numérico. As datas em si, no formato YYYY-MM-DD (ano-mês-dia), podem ser representadas como texto (*character*) ou a própria classe `Date`. Por fim, podemos representar conjuntamente os dados de inflação e as datas armazenando-os em um objeto único do tipo *dataframe*, o qual nada mais é do que uma **tabela** com linhas e colunas. Todos esses objetos fazem parte do ecossistema do R e é através da manipulação destes que tiramos o máximo proveito do *software*.

Enquanto representamos informações do mundo real como diferentes classes no R, um tipo especial de objeto é a função, a qual representa um procedimento preestabelecido que está disponível para o usuário. O R possui uma grande quantidade de funções, as quais possibilitam que o usuário realize uma vasta gama de procedimentos. Por exemplo, os comandos básicos do R, não incluindo demais pacotes, somam um total de 1258 funções. Com base neles e outros iremos importar dados, calcular médias, testar hipóteses, limpar dados, e muito mais.

Cada função possui um próprio nome. Por exemplo, a função `sort()` é um procedimento que ordena valores utilizados como *input*. Caso quiséssemos ordinar os valores `2, 1, 3, 0`, basta inserir no *prompt* o seguinte comando e apertar *enter*:

```
sort(c(2, 1, 3, 0), decreasing = TRUE)
```

```
R> [1] 3 2 1 0
```

O comando `c(2, 1, 3, 0)` combina os valores em um vetor (maiores detalhes sobre comando `c` serão dados em seção futura). Observe que a função `sort` é utilizada com parênteses de início e fim. Esses parênteses servem para destacar as entradas (*inputs*), isto é, as informações enviadas para a função produzir alguma coisa. Observe que cada entrada (ou opção) da função é separada por uma vírgula, tal como em `MinhaFuncao(entrada01, entrada02, entrada03, ...)`. No caso do código anterior, note que usamos a opção `decreasing = TRUE`. Essa é uma instrução específica para a função `sort` ordenar de forma decrescente os elementos do vetor de

entrada. Veja a diferença:

```
sort(c(2, 1, 3, 0), decreasing = FALSE)
```

```
R> [1] 0 1 2 3
```

**O uso de funções está no coração do R** e iremos dedicar grande parte do livro a elas. Por enquanto, essa breve introdução já serve o seu propósito. O principal é entender que uma função usa suas entradas para produzir algo de volta. Nos próximos capítulos iremos utilizar funções já existentes para as mais diferentes finalidades: baixar dados da internet, ler arquivos, realizar testes estatísticos e muito mais. No capítulo 8 iremos tratar deste assunto com maior profundidade, incluindo a forma de escrevermos nossas próprias funções.

## 2.3 O Formato Brasileiro

Antes de começar a explicar o uso do R e RStudio, é importante ressaltar algumas regras de formatação de números e códigos para o caso brasileiro.

**Decimal:** O decimal no R é definido pelo ponto (`.`), tal como em 2.5 e não vírgula, como em 2,5. Esse é o padrão internacional, e a diferença para a notação brasileira gera muita confusão. Alguns softwares, por exemplo o Microsoft Excel, fazem essa conversão automaticamente no momento da importação dos dados. Porém isso não ocorre na maioria dos casos. Como regra geral, utilize vírgulas apenas para separar os termos de entradas em uma função (veja exemplo de seção anterior com função `sort`). Em nenhuma situação deve-se utilizar a vírgula como separador de casas decimais. Mesmo quando estiver exportando dados, sempre dê prioridade para o formato internacional, pois esse será compatível com a grande maioria dos dados e facilitará o uso do *software*.

**Caracteres latinos:** Devido ao seu padrão internacional, o R apresenta problemas para entender caracteres latinos, tal como cedilha e acentos. Caso possa evitar, não utilize esses tipos de caracteres no código para nomeação de variáveis ou arquivos. Nos objetos de classe texto (`character`), é possível utilizá-los desde que a codificação do objeto esteja correta (*UTF-8* ou *Latin1*). Assim, recomenda-se que o código do R seja escrito na língua inglesa. Isso automaticamente elimina o uso de caracteres latinos e facilita a usabilidade do código por outras pessoas que não entendam a língua portuguesa. Destaca-se que essa foi a escolha utilizada para o livro. Os nomes dos objetos nos exemplos estão em inglês, assim como também todos os comentários do código.

**Formato das datas:** Datas no R são formatadas de acordo com a norma ISO 8601, seguindo o padrão YYYY-MM-DD, onde YYYY é o ano em quatro números, MM é o mês e DD é o dia. Por exemplo, uma data em ISO 8601 é 2022-11-23. No Brasil, as datas são formatadas como DD/MM/YYYY. Reforçando a regra, sempre dê preferência

ao padrão internacional. Vale salientar que a conversão entre um formato e outro é bastante fácil e será apresentada em capítulo futuro.

No momento de instalação do R, diversas informações sobre o formato local do seu computador são importadas do seu sistema operacional. Para saber qual o formato que o R está configurado localmente, digite o seguinte comando no *prompt* (canto esquerdo inferior do RStudio) e aperte *enter*:

```
# get local format
Sys.localeconv()
```

```
R>      decimal_point      thousands_sep      grouping
R>      "."                ""                ""
R>      int_curr_symbol    currency_symbol mon_decimal_point
R>      "BRL "            "R$"          ","
R>      mon_thousands_sep mon_grouping    positive_sign
R>      "."                "\003\003"    ""
R>      negative_sign      int_frac_digits frac_digits
R>      "-"                "2"           "2"
R>      p_cs_precedes      p_sep_by_space  n_cs_precedes
R>      "1"                "1"           "1"
R>      n_sep_by_space      p_sign_posn     n_sign_posn
R>      "1"                "1"           "1"
```

A saída de `Sys.localeconv()` mostra como o R interpreta pontos decimais e o separador de milhares, entre outras coisas. Como você pode ver no resultado anterior, este livro foi compilado usando a notação brasileira de moeda (BRL/R\$), mas usa a formatação internacional – o ponto (.) – para decimais.



Muito cuidado ao modificar o formato que o R interpreta os diferentes símbolos e notações. Como regra de bolso, caso precisar usar algum formato específico, faça-o isoladamente dentro do contexto do código. Evite mudanças permanentes pois nunca se sabe onde tais formatos estão sendo usados. Evite, assim, surpresas desagradáveis no futuro.

## 2.4 Tipos de Arquivos

Assim como outros programas, o R possui um ecossistema de arquivos e cada extensão tem uma finalidade diferente. A seguir apresenta-se uma descrição de diversas extensões de arquivos. Os itens da lista estão ordenados por ordem de importância e uso. Note que omitimos arquivos de figuras tal como *.png*, *.jpg*, *.gif* entre outros, pois estes não são exclusivos do R.

**Arquivos com extensão *.R*:** Representam arquivos do tipo texto contendo diver-

sas instruções para o R. Esses são os arquivos que conterão o código da pesquisa e onde passaremos a maior parte do tempo. Também pode ser chamado de um *script* ou rotina de pesquisa. Como sugestão, pode-se dividir toda uma pesquisa em etapas e, para cada, nomear *script* correspondente. Exemplos: *01-Get-Data.R*, *02-Clean-data.R*, *03\_Estimate\_Models.R*.

**Arquivos com extensão *.RData* e *.rds*:** armazenam dados nativos do R. Esses arquivos servem para salvar objetos do R em um arquivo no disco rígido do computador para, em sessão futura, serem novamente carregados. Por exemplo, podes guardar o resultado de uma pesquisa em uma tabela, a qual é salva em um arquivo com extensão *.RData* ou *.rds*. Exemplos: *Raw-Data.RData*, *Table-Results.rds*.

**Arquivos com extensão *.Rmd*, *.md* e *.Rnw*:** São arquivos relacionados a tecnologia *Rmarkdown*. O uso desses arquivos permite a criação de documentos onde texto e código são integrados.

**Arquivos com extensão *.Rproj*:** Contém informações para a edição de projetos no RStudio. O sistema de projetos do RStudio permite a configuração customizada do projeto e também facilita a utilização de ferramentas de controle de código, tal como controle de versões. O seu uso, porém, não é essencial. Para aqueles com interesse em conhecer esta funcionalidade, sugiro a leitura do manual do RStudio<sup>2</sup>. Uma maneira simples de entender os tipos de projetos disponíveis é, no RStudio, clicar em “File”, “New project”, “New Folder” e assim deve aparecer uma tela com todos os tipos possíveis de projetos no RStudio. Exemplo: *My-Dissertation-Project.Rproj*.

## 2.5 Explicando a Tela do RStudio

Após instalar os dois programas, R e RStudio, Procure o ícone do RStudio na área de trabalho ou via menu *Iniciar*. Note que a instalação do R inclui um programa de interface e isso muitas vezes gera confusão. Verifique que estás utilizando o *software* correto. A janela resultante deve ser igual a figura 2.1, apresentada a seguir.

Observe que o RStudio automaticamente detectou a instalação do R e inicializou a sua tela no lado esquerdo. Caso não visualizar uma tela parecida ou chegar em uma mensagem de erro indicando que o R não foi encontrado, repita os passos de instalação do capítulo anterior (seção 1.4).

Como um primeiro exercício, clique em *File*, *New File* e *R Script*. Após, um editor de texto deve aparecer no lado esquerdo da tela do RStudio. É nesse editor que iremos inserir os nossos comandos, os quais são executados de cima para baixo, na mesma direção em que normalmente o lemos. Note que essa direção de execução introduz uma dinâmica de recursividade: cada comando depende do comando executado nas

---

<sup>2</sup><https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>

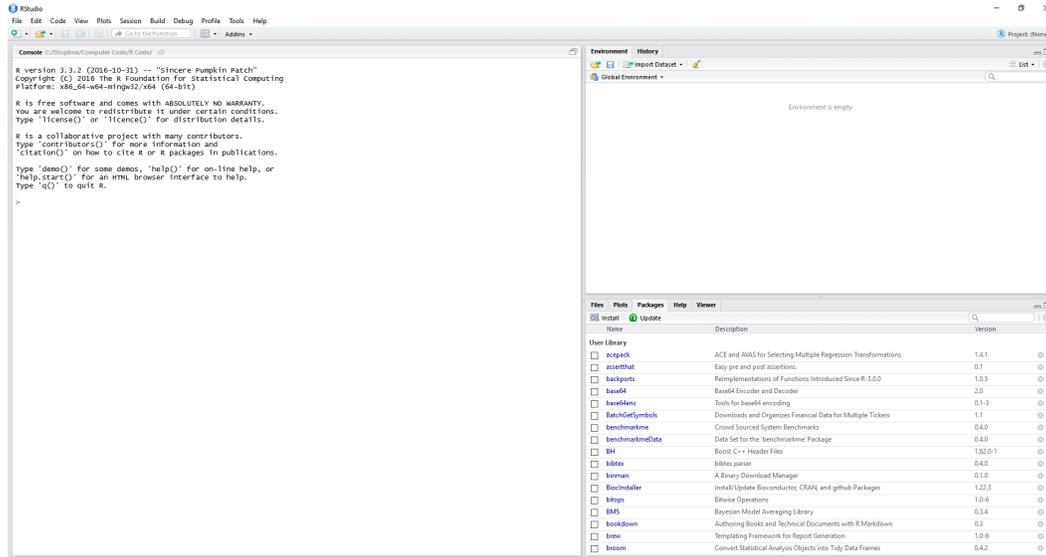


Figura 2.1: A tela do RStudio

linhas anteriores. Após realizar os passos definidos anteriormente, a tela resultante deve ser semelhante à apresentada na figura 2.2.

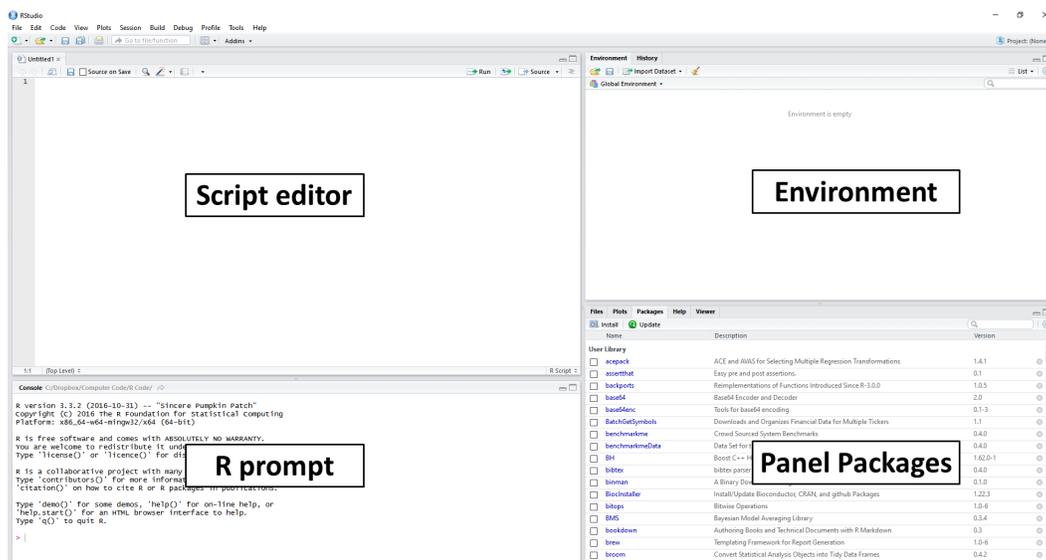


Figura 2.2: Explicando a tela do RStudio



Uma sugestão importante aqui é modificar o esquema de cores do RStudio para uma configuração de **tela escura**. Não é somente uma questão estética mas sim de prevenção e melhoria de sua saúde física. Possivelmente irás passar demasiado tempo na frente do computador. Assim, vale a pena modificar as cores da interface para aliviar seus olhos do constante brilho da tela. Dessa forma, conseguirás trabalhar por mais tempo, sem forçar a sua visão. Podes configurar o esquema de cores do RStudio indo na opção *Tools, Global Options* e então em *Appearance*. Um esquema de cores escuras que pessoalmente gosto e sugiro é o *Ambience*.

Após os passos anteriores, a tela do RStudio deve estar semelhante a Figura 2.2, com os seguintes itens/painéis:

**Editor de scripts (*Script editor*):** localizado no lado esquerdo e acima da tela. Esse painel é utilizado para escrever código e é onde passaremos a maior parte do tempo.

**Console do R (*R prompt*):** localizado no lado esquerdo e abaixo do editor de *scripts*. Apresenta o *prompt* do R, o qual também pode ser utilizado para executar comandos. A principal função do *prompt* é testar código e apresentar os resultados dos comandos inseridos no editor de *scripts*.

**Área de trabalho (*Environment*):** localizado no lado direito e superior da tela. Mostra todos os objetos, incluindo variáveis e funções atualmente disponíveis para o usuário. Observe também a presença do painel *History*, o qual mostra o histórico dos comandos já executados.

**Pacotes (*Panel Packages*):** mostra os pacotes instalados e carregados pelo R. Um pacote é nada mais que um módulo no R, cada qual com sua finalidade específica. Observe a presença de quatro abas: *Files*, para carregar e visualizar arquivos do sistema; *Plots*, para visualizar figuras; *Help*, para acessar o sistema de ajuda do R e *Viewer*, para mostrar resultados dinâmicos e interativos, tal como uma página da internet.

Como um exercício introdutório, vamos inicializar duas variáveis. Dentro do console do R (lado esquerdo inferior), digite os seguintes comandos e aperte *enter* ao final de cada linha. O símbolo `<-` é nada mais que a junção de `<` com `-`. O símbolo `'` representa uma aspa simples e sua localização no teclado Brasileiro é no botão abaixo do *escape* (*esc*), lado esquerdo superior do teclado.

```
# set x and y
x <- 1
y <- 'my text'
```

Após a execução, dois objetos devem aparecer no painel *Environment*, um chamado *x* com o valor 1, e outro chamado *y* com o conjunto de caracteres 'my text'. O histórico de comandos na aba *History* também foi atualizado com os comandos utilizados anteriormente.

Agora, vamos mostrar na tela os valores de *x*. Para isso, digite o seguinte comando no *prompt* e aperte *enter* novamente:

```
# print x
print(x)
```

```
R> [1] 1
```

A função `print` é uma das principais funções para mostrarmos valores no *prompt* do R. O texto apresentado como [1] indica o índice do primeiro número da linha. Para verificar isso, digite o seguinte comando, o qual irá mostrar vários números na tela:

```
# print vector from 50 to 100
print(50:100)
```

```
R> [1] 50 51 52 53 54 55 56 57 58 59 60 61 62 63
R> [15] 64 65 66 67 68 69 70 71 72 73 74 75 76 77
R> [29] 78 79 80 81 82 83 84 85 86 87 88 89 90 91
R> [43] 92 93 94 95 96 97 98 99 100
```

Nesse caso, utilizamos o símbolo `:` em `50:100` para criar uma sequência iniciando em 50 e terminando em 100. Observe que temos valores encapsulados por colchetes (`[]`) no lado esquerda da tela. Esses representam os índices do primeiro elemento apresentado na linha. Por exemplo, o décimo quinto elemento do vetor criado é o valor 64.

## 2.6 Pacotes do R

Um dos grandes benefícios do uso do R é o seu acervo de pacotes. Esses representam um conjunto de procedimentos agrupados em uma coleção de funções e voltados para a resolução de um problema qualquer. O R tem em sua essência uma filosofia de colaboração. Usuários disponibilizam os seus códigos para outras pessoas utilizarem. E, mais importante, **todos os pacotes são gratuitos**, assim como o R. Por exemplo, considere um caso em que está interessado em baixar dados da internet sobre o desemprego histórico no Brasil. Para isso, basta procurar e instalar o pacote específico que realiza esse procedimento.

Esses pacotes podem ser instalados de diferentes fontes, com as principais sendo **CRAN** (*The Comprehensive R Archive Network*) e **Github**. A cada dia aumenta a quantidade e diversidade de pacotes existentes para o R. O próprio autor deste

livro possui diversos pacotes disponíveis no CRAN, cada um para resolver algum problema diferente. Na grande maioria, são pacotes para importar e organizar dados financeiros.

O CRAN é o repositório oficial do R e é livre. Qualquer pessoa pode enviar um pacote e todo código enviado está disponível na internet. Existe, porém, um processo de avaliação que o código passa e certas normas rígidas devem ser respeitadas sobre o formato do código, o manual do usuário e a forma de atualização do pacote. Para quem tiver interesse, um tutorial claro e fácil de seguir é apresentado no site <http://r-pkgs.had.co.nz/intro.html>. As regras completas estão disponíveis no site do CRAN - <https://cran.r-project.org/web/packages/policies.html>. A adequação do código a essas normas é responsabilidade do desenvolvedor e gera um trabalho significativo, principalmente na primeira submissão.

A lista completa de pacotes disponíveis no CRAN, juntamente com uma breve descrição, pode ser acessada no link *packages* do site do R - <https://cran.r-project.org/>. Uma maneira prática de verificar a existência de um pacote para um procedimento específico é carregar a página anterior e procurar no seu navegador de internet a palavra-chave que define o seu procedimento. Caso existir o pacote com a palavra-chave, a procura acusará o encontro do termo na descrição do pacote.

Outra fonte importante para o encontro de pacotes é o *Task Views*, em que são destacados os principais pacotes de acordo com a área e o tipo de uso. Veja a tela do *Task Views* na Figura 2.3.



CRAN Task Views

<a href="#">Bayesian</a>	Bayesian Inference
<a href="#">ChemPhys</a>	Chemometrics and Computational Physics
<a href="#">ClinicalTrials</a>	Clinical Trial Design, Monitoring, and Analysis
<a href="#">Cluster</a>	Cluster Analysis & Finite Mixture Models
<a href="#">DifferentialEquations</a>	Differential Equations
<a href="#">Distributions</a>	Probability Distributions
<a href="#">Econometrics</a>	Econometrics
<a href="#">Environmetrics</a>	Analysis of Ecological and Environmental Data
<a href="#">ExperimentalDesign</a>	Design of Experiments (DoE) & Analysis of Experimental Data
<a href="#">ExtremeValue</a>	Extreme Value Analysis
<a href="#">Finance</a>	Empirical Finance
<a href="#">Genetics</a>	Statistical Genetics
<a href="#">Graphics</a>	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
<a href="#">HighPerformanceComputing</a>	High-Performance and Parallel Computing with R
<a href="#">MachineLearning</a>	Machine Learning & Statistical Learning
<a href="#">MedicalImaging</a>	Medical Image Analysis
<a href="#">MetaAnalysis</a>	Meta-Analysis
<a href="#">Multivariate</a>	Multivariate Statistics
<a href="#">NaturalLanguageProcessing</a>	Natural Language Processing
<a href="#">NumericalMathematics</a>	Numerical Mathematics
<a href="#">OfficialStatistics</a>	Official Statistics & Survey Methodology
<a href="#">Optimization</a>	Optimization and Mathematical Programming
<a href="#">Pharmacokinetics</a>	Analysis of Pharmacokinetic Data
<a href="#">Phylogenetics</a>	Phylogenetics, Especially Comparative Methods
<a href="#">Psychometrics</a>	Psychometric Models and Methods
<a href="#">ReproducibleResearch</a>	Reproducible Research
<a href="#">Robust</a>	Robust Statistical Methods

Figura 2.3: Tela do Task Views

Ao contrário do CRAN, o *Github* não possui restrição quanto ao código enviado e, devido a isso, tende a ser escolhido como ambiente de compartilhamento de có-

digo. A responsabilidade de uso, porém, é do próprio usuário. Na prática, é muito comum os desenvolvedores manterem uma versão em desenvolvimento no *GitHub* e outra oficial no CRAN. Quando a versão em desenvolvimento atinge um estágio de maturidade, a mesma é enviada ao CRAN.

O mais interessante no uso de pacotes é que estes podem ser acessados e instalados diretamente no R via a internet. Para saber qual é a quantidade atual de pacotes no CRAN, digite e execute os seguintes comandos no *prompt*:

```
# find current available packages
df_cran_pkgs <- available.packages()

# get size of matrix
n_cran_pkgs <- nrow(df_cran_pkgs)

# print it
print(n_cran_pkgs)
```

```
R> [1] 18877
```

Atualmente, 2022-11-23 10:55:35, existem 18877 pacotes disponíveis nos servidores do CRAN.

Também se pode verificar a quantidade de pacotes localmente instalados com o comando `installed.packages()`:

```
# get number of local (installed) packages
n_local_pkgs <- nrow(installed.packages())

# print it
print(n_local_pkgs)
```

```
R> [1] 531
```

Nesse caso, o computador em que o livro foi escrito possui 531 pacotes do R instalados. Note que, apesar do autor ser um experiente programador do R, apenas uma pequena fração dos pacotes disponíveis no CRAN está sendo usada! A diversidade dos pacotes é gigantesca.

### 2.6.1 Instalando Pacotes do CRAN

Para instalar um pacote, basta utilizar o comando `install.packages`. Como exemplo, vamos instalar um pacote que será utilizado nos capítulos futuros, o `readr`:

```
# install pkg readr
install.packages('readr')
```

Copie e cole este comando no *prompt* e pronto! O R irá baixar os arquivos necessários e instalar o pacote `readr` e suas dependências. Após isto, as funções relativas ao pacote estarão prontas para serem usadas após o carregamento do módulo (detalhes a seguir). Observe que definimos o nome do pacote na instalação como se fosse texto, com o uso das aspas (`"`). Caso o pacote instalado seja dependente de outros pacotes, o R automaticamente instala todos módulos faltantes. Assim, todos os requerimentos para o uso do respectivo pacote já serão satisfeitos e tudo funcionará perfeitamente. É possível, porém, que um pacote tenha uma dependência externa. Como um exemplo, o pacote `RndTexExams` depende da existência de uma instalação do LaTeX. Geralmente essa é anunciada na sua descrição e um erro é sinalizado na execução do programa quando o LaTeX não é encontrado. Fique atento, portanto, a esses casos.

Aproveitando o tópico, sugiro que o leitor já instale todos os pacotes do `tidyverse` com o seguinte código:

```
# install pkgs from tidyverse
install.packages('tidyverse')
```

O `tidyverse` é um conjunto de pacotes voltados a *data science* e com uma sintaxe própria e consistente, voltada a praticabilidade. Verás que, em uma instalação nova do R, o `tidyverse` depende de uma grande quantidade de pacotes.

### 2.6.2 Instalando Pacotes do Github

Para instalar um pacote diretamente do Github, é necessário instalar antes o pacote `devtools`, disponível no CRAN:

```
# install devtools
install.packages('devtools')
```

Após isto, utilize função `devtools::install_github` para instalar um pacote diretamente do Github. Note que o símbolo `::` indica que função `install_github` pertence ao pacote `devtools`. Com esta particular sintaxe, não precisamos carregar todo o pacote para utilizar apenas uma função.

No exemplo a seguir instalamos a versão em desenvolvimento do pacote `ggplot2`, cuja versão oficial também está disponível no CRAN:

```
# install ggplot2 from github
devtools::install_github("hadley/ggplot2")
```

Observe que o nome do usuário do repositório também é incluído. No caso anterior, o nome *hadley* pertence ao desenvolvedor do `ggplot2`, Hadley Wickham. No decorrer

do livro notará que esse nome aparecerá diversas vezes, dado que Hadley é um prolífico e competente desenvolvedor de diversos pacotes do R e do *tidyverse*.



Um aviso aqui é importante. **Os pacotes do github não são moderados. Qualquer pessoa pode enviar código para lá e o conteúdo não é checado de forma independente.** Nunca instale pacotes do github sem conhecer os autores. Apesar de improvável – nunca aconteceu comigo por exemplo – é possível que esses possuam algum código malicioso.

### 2.6.3 Carregando Pacotes

Dentro de uma rotina de pesquisa, utilizamos a função `library` para carregar um pacote na nossa sessão do R. Ao fecharmos o RStudio ou então iniciar uma nova sessão do R, os pacotes são descarregados. Vale salientar que alguns pacotes, tal como o `base` e o `stats`, são inicializados automaticamente a cada nova sessão. A grande maioria, porém, deve ser carregada no início dos *scripts*. Veja o exemplo a seguir:

```
# load dplyr
library(dplyr)
```

A partir disso, todas as funções do pacote estarão disponíveis para o usuário. Note que não é necessário utilizar aspas (") ao carregar o pacote. Caso utilize uma função específica do pacote e não deseje carregar todo ele, pode fazê-lo através do uso do símbolo especial `::`, conforme o exemplo a seguir.

```
# call fct fortune() from pkg fortunes
fortunes::fortune(10)
```

```
R>
R> Overall, SAS is about 11 years behind R and S-Plus in
R> statistical capabilities (last year it was about 10 years
R> behind) in my estimation.
R> -- Frank Harrell (SAS User, 1969-1991)
R> R-help (September 2003)
```

Nesse caso, utilizamos a função `fortune` do próprio pacote `fortunes`, o qual mostra na tela uma frase possivelmente engraçada escolhida do *mailing list* do R. Nesse caso, selecionamos a mensagem número 10. Se não tiver disponível o pacote, o R mostrará a seguinte mensagem de erro:

```
R> Error in library("fortune") : there is no package called "fortune"
```

Para resolver, utilize o comando `install.packages("fortunes")` para instalar o

pacote no seu computador. Execute o código `fortunes::fortune(10)` no *prompt* para confirmar a instalação. Toda vez que se deparar com essa mensagem de erro, deve instalar o pacote que está faltando.

Outra maneira de carregar um pacote é através da função `require`. Essa tem um comportamento diferente da função `library` e deve ser utilizada dentro da definição de funções ou no teste do carregamento do pacote. Caso o usuário crie uma função customizada que necessite de procedimentos de um pacote em particular, o mesmo deve carregar o pacote no escopo da função. Por exemplo, veja o código a seguir, em que criamos uma função dependente do pacote `quantmod`:

```
my_fct <- function(x){
  require(quantmod)

  df <- getSymbols(x, auto.assign = F)
  return(df)
}
```

Nesse caso, a função `getSymbols` faz parte do pacote `quantmod`. Não se preocupe agora com a estrutura utilizada para criar uma função no R. Essa será explicada em capítulo futuro.



Uma precaução que deve sempre ser tomada quando se carrega um pacote é um possível **conflito de funções**. Por exemplo, existe uma função chamada `filter` no pacote `dplyr` e também no pacote `stats`. Caso carregarmos ambos pacotes e chamarmos a função `filter` no escopo do código, qual delas o R irá usar? Pois bem, a **preferência é sempre para o último pacote carregado**. Esse é um tipo de problema que pode gerar muita confusão. Felizmente, note que o próprio R acusa um conflito de nome de funções no carregamento do pacote. Para testar, inicie uma nova sessão do R e carregue o pacote `dplyr`. Verás que uma mensagem indica haver dois conflitos com o pacote `stats` e quatro com pacote o `base`.

#### 2.6.4 Atualizando Pacotes

Ao longo do tempo, é natural que os pacotes disponibilizados no CRAN sejam atualizados para acomodar novas funcionalidades ou se adaptar a mudanças em suas dependências. Assim, é recomendável que os usuários atualizem os seus pacotes instalados para uma nova versão através da internet. Esse procedimento é bastante fácil. Uma maneira direta de atualizar pacotes é clicar no botão *update* no painel de pacotes no canto direito inferior do RStudio, conforme mostrado na figura 2.4.

A atualização de pacotes através do *prompt* também é possível. Para isso, basta

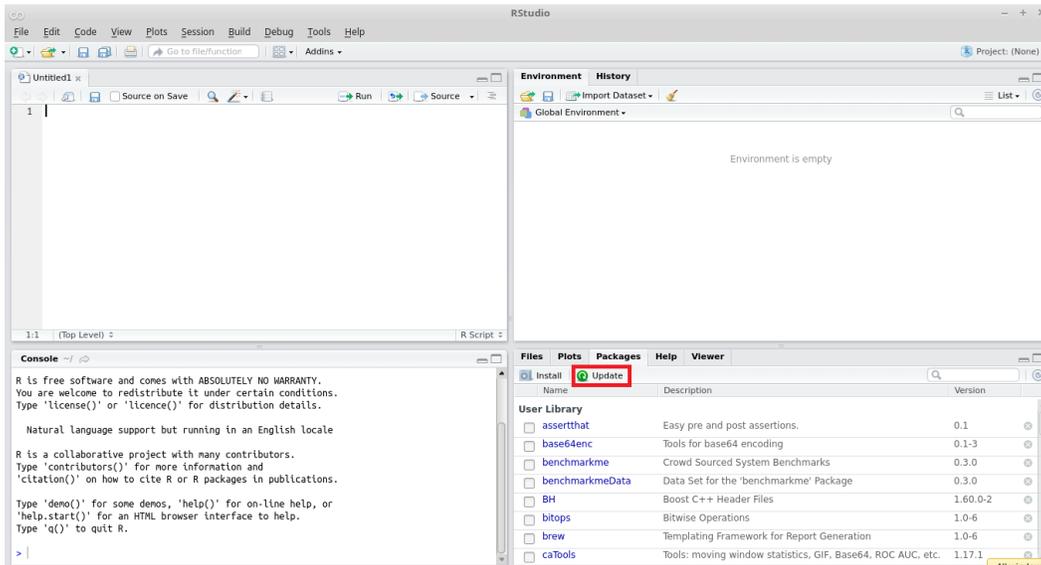


Figura 2.4: Atualizando pacotes no R

utilizar o comando `update.packages`, conforme mostrado a seguir.

```
update.packages()
```

O comando `update.packages()` compara a versão dos pacotes instalados em relação a versão disponível no CRAN. Caso tiver alguma diferença, a nova versão é instalada. Após a execução do comando, todos os pacotes estarão atualizados com a versão disponível nos servidores do CRAN.



Versionamento de pacotes é extremamente importante para manter a reproducibilidade do código. Apesar de ser raro de acontecer, é possível que a atualização de um pacote no R modifique, para os mesmos dados, resultados já obtidos anteriormente. Tenho uma experiência particularmente memorável quando um artigo científico retornou da revisão e, devido a atualização de um dos pacotes, não consegui reproduzir os resultados apresentados no artigo. No final deu tudo certo, mas o trauma fica.

Uma solução para este problema é congelar as versões dos pacotes para cada projeto usando a ferramenta `packrat` do RStudio. Em resumo, o `packrat` faz cópias locais dos pacotes utilizados no projeto, os quais têm preferência aos pacotes do sistema. Assim, se um pacote for atualizado no sistema, mas não no projeto, o código R vai continuar usando a versão mais antiga e seu código sempre rodará nas mesmas condições.

## 2.7 Executando Códigos em um *Script*

Agora, vamos juntar todos os códigos digitados anteriormente e colar na tela do editor (lado esquerdo superior), assim como mostrado a seguir:

```
# set objects
x <- 1
y <- 'my text'

# print it
print(x)
print(1:50)
```

Após colar todos os comandos no editor, salve o arquivo *.R* em alguma pasta pessoal. Esse arquivo, o qual no momento não faz nada de especial, registrou os passos de um algoritmo simples que cria dois objetos e mostra os seus valores. Futuramente esse irá ter mais forma, com a importação de dados, manipulação e modelagem dos mesmos e saída de tabelas e figuras.

No RStudio existem alguns atalhos predefinidos para executar códigos que economizam bastante tempo. Para executar um *script* inteiro, basta apertar **control + shift + s**. Esse é o comando *source*. Com o RStudio aberto, sugiro testar essa combinação de teclas e verificar como o código digitado anteriormente é executado, mostrando os valores no *prompt* do R. Visualmente, o resultado deve ser próximo ao apresentado na figura 2.5.

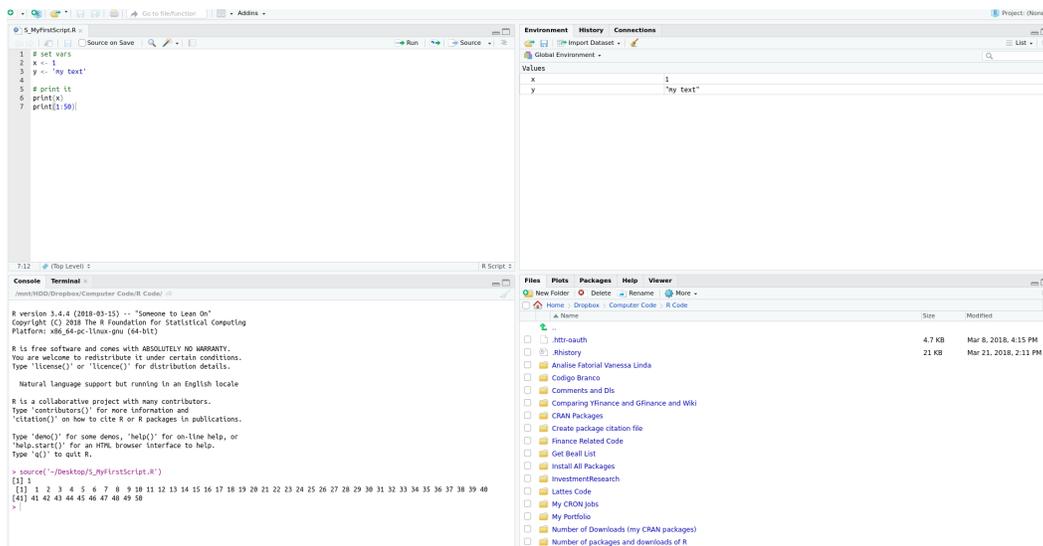


Figura 2.5: Exemplo de Rotina no R

Outro comando muito útil é a execução por linha. Nesse caso não é executado todo o arquivo, mas somente a linha em que o cursor do *mouse* se encontra. Para isto, basta apertar **control+enter**. Esse atalho é bastante útil no desenvolvimento de rotinas pois permite que cada linha seja testada antes de executar o programa inteiro. Como um exemplo de uso, aponte o cursor para a linha `print(x)` e pressione **control + enter**. Verás que o valor de `x` é mostrado na tela do *prompt*. A seguir destaco esses e outros atalhos do RStudio, os quais também são muito úteis.

- **control+shift+s** executa o arquivo atual do RStudio, sem mostrar comandos no *prompt* (sem *eco* – somente saída);
- **control+shift+enter**: executa o arquivo atual, mostrando comandos na tela (com *eco* – código e saída);
- **control+enter**: executa a linha selecionada, mostrando comandos na tela;
- **control+shift+b**: executa os códigos do início do arquivo até a linha atual onde o cursor se encontra;
- **control+shift+e**: executa os códigos da linha onde o cursor se encontra até o final do arquivo.

Sugere-se que esses atalhos sejam memorizados e utilizados. Isso facilita bastante o uso do programa. Para aqueles que gostam de utilizar o *mouse*, uma maneira alternativa para rodar o código do *script* é apertar o botão *source*, localizado no canto direito superior do editor de rotinas. Isto é equivalente ao atalho **control+shift+s**.

Porém, no mundo real de programação, poucos são os casos em que uma análise de dados é realizada por um *script* apenas. Como uma forma de organizar o código, pode-se dividir o trabalho em *N scripts* diferentes, onde um deles é o “mestre”, responsável por rodar os demais.

Neste caso, para executar os *scripts* em sequência, basta chamá-los no *script* mestre com o comando `source`, como no código a seguir:

```
# Import all data
source('01-import-data.R')

# Clean up
source('02-clean-data.R')

# Build tables
source('03-build-table.R')
```

Nesse caso, o código anterior é equivalente a abirmos e executarmos (*control + shift + s*) cada um dos *scripts* sequencialmente.

Como podemos ver, existem diversas maneiras de executar uma rotina de pesquisa. Na prática, porém, iras centralizar o uso em dois comandos apenas: **control+shift+s** para rodar o *script* inteiro e **control+enter** para rodar por

linha.

## 2.8 Testando Código

O desenvolvimento de códigos em R segue um conjunto de etapas. Primeiro você escreverá uma nova linha de comando em uma rotina. Essa linha será testada com o atalho `control + enter`, verificando-se a ocorrência de erros e as saídas na tela. Caso não houver erro e o resultado for igual ao esperado, parte-se para a próxima linha de código.

Um ciclo de trabalho fica claro, a escrita do código da linha atual é seguida pela execução, seguido da verificação de resultados, modificação caso necessário e assim por diante. Esse é um processo normal e esperado. Dado que uma rotina é lida e executada de cima para baixo, você precisa ter certeza de que cada linha de código está corretamente especificada antes de passar para a próxima.

Quando você está tentando encontrar um erro em um *script* preexistente, o R oferece algumas ferramentas para controlar e avaliar sua execução. Isso é especialmente útil quando você possui um código longo e complicado. A ferramenta de teste mais simples e fácil de utilizar que o RStudio oferece é o ponto de interrupção do código. No RStudio, você pode clicar no lado esquerdo do editor e aparecerá um círculo vermelho, como na Figura 2.6.

```
1 # set x
2 x <- 1
3
4 # set y
5 ● y <- 'My humble text'
6
7 # print contents of x
8 print(x)|
```

Figura 2.6: Exemplo de debug

O círculo vermelho indica um ponto de interrupção do código que forçará o R a pausar a execução nessa linha. Quando a execução atinge o ponto de interrupção, o *prompt* mudará para `browser[1]>` e você poderá verificar o conteúdo dos objetos. No console, você tem a opção de continuar a execução para o próximo ponto de interrupção ou interrompê-la. O mesmo resultado pode ser alcançado usando a função `browser`. Dê uma olhada:

```
# set x
x <- 1

# set y
browser()
y <- 'My humble text'

# print contents of x
print(x)
```

O resultado prático do código anterior é o mesmo que utilizar o círculo vermelho do RStudio, figura 2.6. Porém, o uso do `browser` permite mais controle sobre onde a execução deve ser pausada. Como um teste, copie e cole o código anterior no RStudio, salve em um novo *script* e execute com *Control + Shift + S*. Para sair do ambiente de depuração (*debug*), aperte *enter* no *prompt* do RStudio.

## 2.9 Criando Objetos Simples

Um dos comandos mais básicos no R é a definição de objetos. Como foi mostrado nas seções anteriores, pode-se definir um objeto com o uso do comando `<-`, o qual, para o português, é traduzido para o verbo *defina* (*assign* em inglês). Considere o seguinte código:

```
# set x
my_x <- 123

# set x, y and z in one line
my_x <- 1 ; my_y <- 2; my_z <- 3
```

Lê-se esse código como *x é definido como 123*. A direção da seta define onde o valor será armazenado. Por exemplo, utilizar `123 -> my_x` também funcionaria, apesar de ser uma sintaxe pouco utilizada ou recomendada. Note que também é possível escrever diversos comandos na mesma linha com o uso da semi-vírgula (;).



O uso do símbolo `<-` para a definição de objetos é específico do R. Na época da concepção da linguagem *S*, de onde o R foi baseado, existiam teclados com uma tecla específica que definia diretamente o símbolo de seta. Teclados contemporâneos, porém, não possuem mais esta configuração. Uma alternativa é utilizar o atalho para o símbolo, o qual, no Windows, é definido por `alt + -`.

É possível também usar o símbolo `=` para definir objetos assim como o `<-`. Saliento que esta é prática comum em outras linguagens de programação. Porém, no ecossistema do R, a utilização do `=` com esse fim específico não é recomendada. O símbolo de igualdade tem o seu uso especial e resguardado na definição de argumentos de uma função tal como `sort(x = 1:10, decreasing = TRUE)`.

O nome dos objetos é importante no R. Tirando alguns casos específicos, o usuário pode nomear os objetos como quiser. Essa liberdade, porém, pode ser um problema. É desejável sempre dar nomes curtos que façam sentido ao conteúdo do objeto e que sejam simples de entender. Isso facilita o entendimento do código por outros usuários e faz parte das normas sugeridas para a estruturação do código. Observe que todos os objetos criados nesse livro possuem nomenclatura em inglês e formatação específica, onde espaços entre substantivos e verbos são substituídos por traço baixo, tal como em `my_x` e `my_csv_file`. Aqui, o mais importante é a consistência do formato. Sempre mantenha o mesmo padrão em todo o código. No capítulo 13 vamos ir mais a fundo nesta questão de estrutura de código.

O R executa o código procurando objetos e funções disponíveis no seu ambiente de trabalho (*environment*). Se tentarmos acessar um objeto que não existe, o R irá retornar uma mensagem de erro:

```
print(z)
```

```
R> Error in print(z): object 'z' not found
```

Isso ocorre pois o objeto `z` não existe na sessão atual do R. Se criarmos uma variável `z` como `z <- 123` e repetirmos o comando `print(z)`, não teremos a mesma mensagem de erro.

Um ponto importante aqui é a definição de objetos de classes diferentes com o uso de símbolos específicos. O uso de aspas duplas (`" "`) ou simples (`' '`) define objetos da classe texto enquanto números são definidos pelo próprio valor. Conforme será mostrado, cada objeto no R tem uma classe e cada classe tem um comportamento diferente. Portanto, objetos criados com o uso de aspas pertencem à classe *character*. Podemos confirmar isso via código:

```
# set vars
x <- 1
y <- '1'

# display classes
class(x)
```

```
R> [1] "numeric"
```

```
class(y)
```

```
R> [1] "character"
```

As saídas anteriores mostram que a variável `x` é do tipo numérico, enquanto a variável `y` é do tipo texto (*character*). Ambas fazem parte das classes básicas de objetos no R. Por enquanto, este é o mínimo que debes saber para avançar nos próximos capítulos. Iremos estudar este assunto mais profundamente no capítulo 7.

## 2.10 Criando Vetores

Nos exemplos anteriores criamos objetos simples tal como `x <- 1` e `x <- 'abc'`. Enquanto isso é suficiente para demonstrar os comandos básicos do R, na prática tais comandos são bastante limitados, uma vez que um problema real de análise de dados certamente irá ter um maior volume de informações do mundo real.

Um dos procedimentos mais utilizados no R é a criação de vetores atômicos. Esses são objetos que guardam uma série de elementos. Todos os elementos de um vetor atômico devem possuir a mesma classe, o que justifica a sua propriedade *atômica*. Um exemplo seria representar no R uma série de preços diários de uma ação. Tal série possui vários valores numéricos que formam um vetor da classe numérica.

**Vetores atômicos são criados no R através do uso do comando `c`, o qual é oriundo do verbo em inglês *combine*.** Por exemplo, caso eu quisesse *combinar* os valores 1, 2 e 3 em um objeto/vetor, eu poderia fazê-lo através do seguinte comando:

```
# set vector
x <- c(1, 2, 3)

# print it
print(x)
```

```
R> [1] 1 2 3
```

Esse comando funciona da mesma maneira para qualquer número de elementos. Caso necessário, poderíamos criar um vetor com mais elementos simplesmente adicionando valores após o 3, tal como em `x <- c(1, 2, 3, 4, 5)`.

O uso do comando `c` não é exclusivo para vetores numéricos. Por exemplo, poderíamos criar um vetor de outra classe de dados, tal como *character*:

```
y <- c('text 1', 'text 2', 'text 3', 'text 4')
print(y)
```

```
R> [1] "text 1" "text 2" "text 3" "text 4"
```

A única restrição no uso do comando `c` é que todos os itens do vetor **tenham a mesma classe**. Se inserirmos dados de classes diferentes, o R irá tentar transformar os itens para a mesma classe seguindo uma lógica própria, onde a classe mais complexa sempre tem preferência. Caso ele não consiga transformar todos os elementos para uma classe só, uma mensagem de erro será retornada. Observe no próximo exemplo como os valores numéricos no primeiro e segundo elemento de `x` são transformados para a classe de caracteres.

```
# numeric class
x <- c(1, 2)
class(x)
```

```
R> [1] "numeric"
```

```
# character class
x <- c(1, 2, '3')
class(x)
```

```
R> [1] "character"
```

Outra utilização do comando `c` é a combinação de vetores. De fato, isto é exatamente o que fizemos ao executar o código `c(1, 2, 3)`. Neste caso, cada vetor possuía um elemento. Podemos realizar o mesmo com vetores maiores. Veja a seguir:

```
# set x and y
x <- c(1, 2, 3)
y <- c(4, 5)

# print concatenation between x and y
print(c(x, y))
```

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

Portanto, o comando `c` possui duas funções principais: criar e combinar vetores.

## 2.11 Conhecendo os Objetos Criados

Após a execução de diversos comandos no editor ou *prompt*, é desejável saber quais são os objetos criados pelo código. É possível descobrir essa informação simplesmente olhando para o lado direito superior do RStudio, na aba da área de trabalho. Porém, existe um comando que sinaliza a mesma informação no *prompt*. Com o fim de saber quais são as variáveis atualmente disponíveis na memória do R, pode-se utilizar o comando `ls`. Observe o exemplo a seguir:

```
# set vars
x <- 1
```

```
y <- 2
z <- 3

# show current objects
ls()
```

```
R> [1] "x" "y" "z"
```

Os objetos `x`, `y` e `z` foram criados e estavam disponíveis no ambiente de trabalho atual, juntamente com outros objetos. Para descobrir os valores dos mesmos, basta digitar os nomes dos objetos e apertar **enter** no *prompt*:

```
x
```

```
R> [1] 1
```

```
y
```

```
R> [1] 2
```

```
z
```

```
R> [1] 3
```

Destaca-se que digitar o nome do objeto na tela tem o mesmo resultado que utilizar a função `print`. De fato, ao executar o nome de uma variável, internamente o R passa esse objeto para a função `print`.

No R, conforme já mostrado, todos os objetos pertencem a alguma classe. Para descobrir a classe de um objeto, basta utilizar a função `class`. Observe no exemplo a seguir que `x` é um objeto da classe numérica e `y` é um objeto da classe de texto (*character*).

```
# set vars
x <- 1
y <- 'a'

# check classes
class(x)
```

```
R> [1] "numeric"
```

```
class(y)
```

```
R> [1] "character"
```

Outra maneira de conhecer melhor um objeto é verificar a sua representação em texto. Todo objeto no R possui uma representação textual e a verificação desta é realizada através da função `str`:

```
# print textual representation of a vector
x <- 1:10
print(str(x))
```

```
R> int [1:10] 1 2 3 4 5 6 7 8 9 10
R> NULL
```

Essa função é particularmente útil quando se está tentando entender os detalhes de um objeto mais complexo, tal como uma tabela. A utilidade da representação textual é que nela aparece o tamanho do objeto e suas classes internas. Nesse caso, o objeto `x` é da classe *integer* e possui dez elementos.

## 2.12 Mostrando e Formatando Informações na Tela

Como já vimos, é possível mostrar o valor de uma variável na tela de duas formas, digitando o nome dela no *prompt* ou então utilizando a função `print`. Explicando melhor, a função `print` é voltada para a apresentação de objetos e pode ser customizada para qualquer tipo. Por exemplo, caso tivéssemos um objeto de classe chamada `MyTable` que representasse um objeto tabular, poderíamos criar uma função chamada `print.MyTable` que irá mostrar uma tabela na tela com um formato especial tal como número de linhas, nomes das colunas, etc. A função `print`, portanto, pode ser customizada para cada classe de objeto.

Porém, existem outras funções específicas para apresentar texto (e não objetos) no *prompt*. A principal delas é `message`. Essa toma como *input* um texto, processa-o para símbolos específicos e o apresenta na tela. Essa função é muito mais poderosa e personalizável do que `print`.

Por exemplo, caso quiséssemos mostrar na tela o texto 'O valor de `x` é igual a 2', poderíamos fazê-lo da seguinte forma:

```
# set var
x <- 2

# print with message()
message('The value of x is', x)
```

```
R> The value of x is2
```

Função `message` também funciona para vetores:

```
# set vec
x <- 2:5

# print with message()
```

```
message('The values in x are: ', x)
```

```
R> The values in x are: 2345
```

A customização da saída da tela é possível através de comandos específicos. Por exemplo, se quiséssemos quebrar a linha da tela, poderíamos fazê-lo através do uso do caractere reservado `\n`:

```
# set char
my_text <- 'First line,\nSecond Line,\nThird Line'

# print with new lines
message(my_text)
```

```
R> First line,
R> Second Line,
R> Third Line
```

Observe que o uso do `print` não resultaria no mesmo efeito, uma vez que esse comando apresenta o texto como ele é, sem processar para efeitos específicos:

```
print(my_text)
```

```
R> [1] "First line,\nSecond Line,\nThird Line"
```

Outro exemplo no uso de comandos específicos para texto é adicionar um espaçamento *tab* no texto apresentado com o símbolo `\t`. Veja a seguir:

```
# set char with \t
my_text_1 <- 'A and B'
my_text_2 <- '\tA and B'
my_text_3 <- '\t\tA and B'

# print with message()
message(my_text_1)
```

```
R> A and B
```

```
message(my_text_2)
```

```
R> A and B
```

```
message(my_text_3)
```

```
R> A and B
```

Vale destacar que, na grande maioria dos casos de pesquisa, será necessário apenas o uso de `\n` para formatar textos de saída. Outras maneiras de manipular a saída

de texto no *prompt* com base em símbolos específicos são encontradas no manual oficial do R.

Parte do processo de apresentação de texto na tela é a customização do mesmo. Para isto, existem duas funções muito úteis: `paste` e `format`.

A função `paste` *cola* uma série de caracteres juntos. É uma função muito útil, a qual será utilizada intensamente para o resto dos exemplos deste livro. Observe o código a seguir:

```
# set chars
my_text_1 <- 'I am a text'
my_text_2 <- 'very beautiful'
my_text_3 <- 'and informative.'

# using paste and message
message(paste(my_text_1, my_text_2, my_text_3))
```

```
R> I am a text very beautiful and informative.
```

O resultado anterior não está muito longe do que fizemos no exemplo com a função `print`. Note, porém, que a função `paste` adiciona um espaço entre cada texto. Caso não quiséssemos esse espaço, poderíamos usar a função `paste0`:

```
# using paste0
message(paste0(my_text_1, my_text_2, my_text_3))
```

```
R> I am a textvery beautifuland informative.
```



Uma alternativa a função `message` é `cat` (*concatenate and print*). Não é incomum encontrarmos códigos onde mensagens para o usuário são transmitidas via `cat` e não `message`. Como regra, dê preferência a `message` pois esta é mais fácil de controlar. Por exemplo, caso o usuário quiser silenciar uma função, omitindo todas saídas da tela, bastaria usar o comando `suppressMessages`.

Outra possibilidade muito útil no uso do `paste` é modificar o texto entre a junção dos itens a serem colados. Por exemplo, caso quiséssemos adicionar uma vírgula e espaço (,) entre cada item, poderíamos fazer isso através do uso do argumento `sep`, como a seguir:

```
# using custom separator
message(paste(my_text_1, my_text_2, my_text_3, sep = ', '))
```

```
R> I am a text, very beautiful, and informative.
```

Caso tivéssemos um vetor atômico com os elementos da frase em um objeto apenas, poderíamos atingir o mesmo resultado utilizando `paste` o argumento `collapse`:

```
# using paste with collapse argument
my_text <-c('Eu sou um texto', 'muito bonito', 'e charmoso.')
message(paste(my_text, collapse = ', '))
```

```
R> Eu sou um texto, muito bonito, e charmoso.
```

Prosseguindo, o comando `format` é utilizado para formatar números e datas. É especialmente útil quando formos montar tabelas e buscarmos apresentar os números de uma maneira visualmente atraente. Por definição, o R apresenta uma série de dígitos após a vírgula:

```
# message without formatting
message(1/3)
```

```
R> 0.3333333333333333
```

Caso quiséssemos apenas dois dígitos aparecendo na tela, utilizaríamos o seguinte código:

```
# message with format and two digits
message(format(1/3, digits=2))
```

```
R> 0.33
```

Tal como, também é possível mudar o símbolo de decimal:

```
# message with format and two digits
message(format(1/3, decimal.mark = ','))
```

```
R> 0,3333333
```

Tal flexibilidade é muito útil quando devemos reportar resultados respeitando algum formato local tal como o Brasileiro.

Uma alternativa recente e muito interessante para o comando `base::paste` é `stringr::str_c` e `stringr::str_glue`. Enquanto a primeira é quase idêntica a `paste0`, a segunda tem uma maneira peculiar de juntar objetos. Veja um exemplo a seguir:

```
library(stringr)

# define some vars
my_name <- 'Pedro'
my_age <- 23
```

```
# using base::paste0
my_str_1 <- paste0('My name is ', my_name, ' and my age is ', my_age)

# using stringr::str_c
my_str_2 <- str_c('My name is ', my_name, ' and my age is ', my_age)

# using stringr::str_glue
my_str_3 <- str_glue('My name is {my_name} and my age is {my_age}')

identical(my_str_1, my_str_2)
```

```
R> [1] TRUE
```

```
identical(my_str_1, my_str_3)
```

```
R> [1] FALSE
```

```
identical(my_str_2, my_str_3)
```

```
R> [1] FALSE
```

Como vemos, temos três alternativas para o mesmo resultado final. Note que `str_glue` usa de chaves para definir as variáveis dentro do próprio texto. Esse é um formato muito interessante e prático.

## 2.13 Conhecendo o Tamanho dos Objetos

Na prática de programação com o R, é muito importante saber o tamanho das variáveis que estão sendo utilizadas. Isso serve não somente para auxiliar o usuário na verificação de possíveis erros do código, mas também para saber o tamanho necessário em certos procedimentos de iteração tal como *loops*, os quais serão tratados em capítulo futuro.

No R, o tamanho do objeto pode ser verificado com o uso de quatro principais funções: `length`, `nrow`, `ncol` e `dim`.

A função `length` é destinada a objetos com uma única dimensão, tal como vetores atômicos:

```
# set x
x <- c(2,3,3,4,2,1)

# get length x
n <- length(x)
```

```
# display message
message(paste('The length of x is', n))
```

```
R> The length of x is 6
```

Para objetos com mais de uma dimensão, por exemplo matrizes e dataframes, utilizam-se as funções `nrow`, `ncol` e `dim` para descobrir o número de linhas (primeira dimensão) e o número de colunas (segunda dimensão). Veja a diferença a seguir.

```
# set matrix and print it
x <- matrix(1:20, nrow = 4, ncol = 5)
print(x)
```

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

```
# find number of rows, columns and elements
my_nrow <- nrow(x)
my_ncol <- ncol(x)
my_length <- length(x)

# print message
message(paste('\nThe number of lines in x is ', my_nrow))
```

```
R>
R> The number of lines in x is 4
```

```
message(paste('\nThe number of columns in x is ', my_ncol))
```

```
R>
R> The number of columns in x is 5
```

```
message(paste('\nThe number of elements in x is ', my_length))
```

```
R>
R> The number of elements in x is 20
```

Já a função `dim` mostra a dimensão do objeto, resultando em um vetor numérico como saída. Essa deve ser utilizada quando o objeto tiver mais de duas dimensões. Na prática, esses casos são raros. Um exemplo para a variável `x` é dado a seguir:

```
print(dim(x))
```

```
R> [1] 4 5
```

Para o caso de objetos com mais de duas dimensões, podemos utilizar a função `array` para criá-los e `dim` para descobrir o seu tamanho:

```
# set array with dimension
my_array <- array(1:9, dim = c(3,3,3))

# print it
print(my_array)
```

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

```
# print its dimension
print(dim(my_array))
```

```
R> [1] 3 3 3
```

Uma observação importante aqui é que as funções `length`, `nrow`, `ncol` e `dim` não servem para descobrir o número de letras em um texto. Esse é um erro bastante comum. Por exemplo, caso tivéssemos um objeto do tipo texto e usássemos a função `length`, o resultado seria o seguinte:

```
# set char object
my_char <- 'abcde'
```

```
# find its length (and NOT number of characters)
print(length(my_char))
```

```
R> [1] 1
```

Isso ocorre pois a função `length` retorna o número de elementos. Nesse caso, `my_char` possui apenas um elemento. Para descobrir o número de caracteres no objeto, utilizamos a função `nchar`, conforme a seguir:

```
# using nchar for number of characters
print(nchar(my_char))
```

```
R> [1] 5
```

Reforçando, cada objeto no R tem suas propriedades e funções específicas para manipulação.

## 2.14 Selecionando Elementos de um Vetor Atômico

Após a criação de um vetor atômico de qualquer classe, é possível que se esteja interessado em apenas um ou alguns elementos desse mesmo vetor. Por exemplo, caso estivéssemos buscando atualizar o valor de um portfólio de investimento, o nosso interesse dentro de um vetor contendo preços de uma ação é somente para o preço mais recente. Todos os demais preços não seriam relevantes para a nossa análise e, portanto, poderiam ser ignorados.

Esse processo de seleção de *pedaços* de um vetor atômico é chamado de indexação e é executado através do uso de colchetes `[]`. Observe o exemplo de código a seguir:

```
# set my_x
my_x <- c(1, 5, 4, 3, 2, 7, 3.5, 4.3)
```

Se quiséssemos apenas o terceiro elemento de `my_x`, utilizaríamos o operador de colchete da seguinte forma:

```
# get third element of my_x
elem_x <- my_x[3]
print(elem_x)
```

```
R> [1] 4
```

Também podemos utilizar o comando `length`, apresentado anteriormente, para acessar o último elemento do vetor:

```
# get last element of my_x
last_elem <- my_x[length(my_x)]
```

```
# print it
print(last_elem)
```

```
R> [1] 4.3
```

No caso de estarmos interessado apenas no último e penúltimo valor de `my_x` utilizaríamos o operador de sequência (`:`):

```
# get last and second last elements
piece_x_1 <- my_x[ (length(my_x)-1):length(my_x) ]

# print it
print(piece_x_1)
```

```
R> [1] 3.5 4.3
```

Uma propriedade única da linguagem R é que, caso for acessado uma posição que não existe no vetor, o programa retorna o valor `NA` (*not available*). Veja a seguir, onde tenta-se obter o quarto valor de um vetor com apenas três elementos.

```
# set vec
my_vec <- c(1,2,3)

# find fourth element (NA returned!)
print(my_vec[4])
```

```
R> [1] NA
```

É importante conhecer esse comportamento do R, pois o não tratamento desses erros pode gerar problemas difíceis de identificar em um código mais complexo. Em outras linguagens de programação, a tentativa de acesso a elementos não existentes geralmente retorna um erro e cancela a execução do resto do código. No caso do R, dado que o acesso a elementos inexistentes não gera erro, é possível que isso gere um problema em outras partes do *script*.



Geralmente, a ocorrência de `NA`s (*Not Available*) sugere a existência de problema no código. Saiba que `NA` indicam a falta de dados e são contagiosos: tudo que interagir com objeto do tipo `NA`, seja uma soma ou multiplicação, irá também virar `NA`. **O usuário deve prestar atenção toda vez que surgirem valores `NA` de forma inesperada nos objetos criados.** Uma inspeção nos índices dos vetores pode ser necessária.

O uso de indexadores é muito útil quando se está procurando por itens de um vetor que satisfaçam alguma condição. Por exemplo, caso quiséssemos todos os valores de

`my_x` que são maiores que 3, utilizaríamos o seguinte comando:

```
# get all values higher than 3
piece_x_2 <- my_x[my_x>3]

# print it
print(piece_x_2)
```

```
R> [1] 5.0 4.0 7.0 3.5 4.3
```

É possível também indexar por mais de uma condição através dos operadores de lógica `&` (*e*) e `|` (*ou*). Por exemplo, caso quiséssemos os valores de `my_x` maiores que 2 e menores que 4, usaríamos o seguinte comando:

```
# get all values higher than 2 AND lower than 4
piece_x_3 <- my_x[ (my_x>2) & (my_x<4) ]

# print it
print(piece_x_3)
```

```
R> [1] 3.0 3.5
```

Da mesma forma, havendo interesse nos itens que são menores que 3 ou maiores que 6, teríamos:

```
# get all values lower than 3 OR higher than 6
piece_x_4 <- my_x[ (my_x<3)|(my_x>6) ]

# print it
print(piece_x_4)
```

```
R> [1] 1 2 7
```

A indexação lógica também funciona com a interação de diferentes variáveis, isto é, podemos utilizar uma condição lógica em uma variável para selecionar itens em outra:

```
# set my_x and my_y
my_x <- c(1, 4, 6, 8, 12)
my_y <- c(-2, -3, 4, 10, 14)

# find elements in my_x where my_y are positive
my_piece_x <- my_x[ my_y > 0 ]

# print it
print(my_piece_x)
```

```
R> [1] 6 8 12
```

Olhando mais de perto o processo de indexação, vale salientar que, quando utilizamos uma condição de indexação dos dados, esta-se criando uma variável do tipo lógica. Essa toma apenas dois valores: **TRUE** (verdadeiro) ou **FALSE** (falso). É fácil perceber isso quando criamos o teste lógico em um objeto e o mostramos na tela:

```
# set logical object
my_logical <- my_y > 0

# print it
print(my_logical)
```

```
R> [1] FALSE FALSE TRUE TRUE TRUE
```

```
# show its class
class(my_logical)
```

```
R> [1] "logical"
```

As demais propriedades e operações com vetores lógicos serão explicadas em capítulo futuro.

## 2.15 Limpando a Memória

Após a criação de diversas variáveis, o ambiente de trabalho do R pode ficar cheio de conteúdo já utilizado e dispensável. Nesse caso, é desejável limpar a memória do programa. Geralmente isso é realizado no começo de um *script*, de forma que toda vez que o script for executado, a memória estará totalmente limpa antes de qualquer cálculo. Além de desocupar a memória do computador, isso ajuda a evitar possíveis erros no código. Na grande maioria dos casos, porém, a limpeza do ambiente de trabalho deve ser realizada apenas uma vez.

Por exemplo, dada uma variável **x**, podemos excluí-la da memória com o comando **rm**, conforme mostrado a seguir:

```
# set x and y
x <- 1
y <- 2

# print all existing objects
ls()

# remove x from memory
rm('x')
```

```
# print objects again
ls()
```

Observe que o objeto `x` não estará mais disponível após o uso do comando `rm('x')`.

Entretanto, em situações práticas é desejável limpar toda a memória utilizada por todos os objetos disponíveis no R. Pode-se atingir esse objetivo com o seguinte código:

```
# clean up workspace (all existing objects)
rm(list = ls())
```

O termo `list` é um argumento da função `rm`, o qual define quais objetos serão eliminados. Já o comando `ls()` mostra todos os objetos disponíveis atualmente. Portanto, ao encadear ambos os comandos, limpamos da memória **todos** os objetos disponíveis para o R. Como comentado, uma boa política de programação é sempre iniciar o *script* limpando a memória do R. .



A limpeza da memória em *scripts* é uma estratégia controversa. Alguns autores argumentam que é melhor não limpar a memória pois isso pode apagar resultados importantes. Na minha opinião, acho fundamental limpar a memória, desde que todos resultados sejam reproduzíveis. Ao iniciar um código sempre do mesmo estado, isto é, nenhuma variável criada, fica mais fácil de entender e capturar possíveis *bugs*.

## 2.16 Mostrando e Mudando o Diretório de Trabalho

Assim como outros softwares, **o R sempre trabalha em algum diretório**. É com base nesse diretório que o R procura arquivos para importar dados. É nesse mesmo diretório que o R salva arquivos, caso não definirmos um endereço no computador explicitamente. Essa saída pode ser um arquivo de uma figura, um arquivo de texto ou uma planilha eletrônica. **Como boa prática de criação e organização de *scripts*, deve-se sempre mudar o diretório de trabalho para onde o arquivo do *script* está localizado.**

Em sua inicialização, o R possui como diretório *default* a pasta de documentos do usuário cujo atalho é o tilda (`'~'`).

Para mostrar o diretório atual de trabalho, basta utilizar a função `getwd`:

```
# get current directory
my_dir <- getwd()
```

```
# print it
print(my_dir)
```

```
R> [1] home/msperlin/adfeR/01-Book Content
```

O resultado do código anterior mostra a pasta onde este livro foi escrito. Esse é o diretório onde os arquivos do livro foram compilados dentro do ambiente Linux.

A mudança de diretório de trabalho é realizada através do comando `setwd`. Por exemplo, caso quiséssemos mudar o nosso diretório de trabalho para `C:/Minha pesquisa/`, basta digitar no *prompt*:

```
# set dir
my_d <- 'C:/Minha Pesquisa/'
setwd(my_d)
```

Enquanto para casos simples, como o anterior, lembrar o nome do diretório é fácil, em casos práticos o diretório de trabalho pode ser em um lugar mais profundo da raiz de diretórios do sistema de arquivos. Nessa situação, uma estratégia eficiente para descobrir a pasta de trabalho é utilizar um explorador de arquivos, tal como o *Explorer* no Windows. Abra esse aplicativo e vá até o local onde quer trabalhar com o seu *script*. Após isso, coloque o cursor na barra de endereço e selecione todo o endereço. Aperte *control+c* para copiar o endereço para a área de transferência. Volte para o seu código e cole o mesmo no código. **Atenção nesta etapa, o Windows utiliza a barra invertida para definir endereços no computador, enquanto o R utiliza a barra normal.** Caso tente utilizar a barra invertida, um erro será mostrado na tela. Veja o exemplo a seguir.

```
my_d <- 'C:\Minha pesquisa\'
setwd(my_d)
```

O erro terá a seguinte mensagem:

```
Error: '\M' is an unrecognized escape in character string..."
```

A justificativa para o erro é que a barra invertida `\` é um caractere reservado no R e não pode ser utilizado isoladamente. Caso precises, podes defini-lo no objeto de texto com dupla barra, tal como em `\\`. Veja no exemplo a seguir, onde a dupla barra é substituída por uma barra única:

```
# set char with \
my_char <- 'using \\'

# print it
message(my_char)
```

```
R> using \
```

A solução do problema é simples. Após copiar o endereço, modifique todas as barras para a barra normal, assim como no código a seguir:

```
my_d <- 'C:/Minha pesquisa/'  
setwd(my_d)
```

É possível também utilizar barras invertidas duplas `\\` na definição de diretórios, porém não se recomenda essa formatação, pois não é compatível com outros sistemas operacionais.

Outro ponto importante aqui é o uso de endereços relativos. Por exemplo, caso esteja trabalhando em um diretório que contém um subdiretório chamado `Data`, podes entrar nele com o seguinte código:

```
# change to subfolder  
setwd('Data')
```

Outra possibilidade pouco conhecida no uso de `setwd` é que é possível entrar em níveis inferiores do sistema de diretórios com `..`, tal como em:

```
# change to previous level  
setwd('..')
```

Portanto, caso estejas trabalhando no diretório `C:/My Research/` e executar o comando `setwd('..')`, o diretório atual de trabalho viraria `C:/`, um nível inferior a `C:/My Research/`.

Uma maneira mais moderna e pouco conhecida de definir o diretório de trabalho é usar as funções internas do RStudio. Este é um conjunto de funções que só funcionam dentro do RStudio e fornecem diversas informações sobre o arquivo sendo editado. Para descobrir o caminho do arquivo atual que está sendo editado no RStudio e configurar o diretório de trabalho para lá, você pode escrever:

```
my_path <- dirname(rstudioapi::getActiveDocumentContext())$path  
setwd(my_path)
```

Dessa forma, o *script* mudará o diretório para sua própria localização. Apesar de não ser um código exatamente elegante, ele é bastante funcional. Caso copie o arquivo para outro diretório, o valor de `my_path` muda para o novo diretório. Esteja ciente, no entanto, de que esse truque só funciona no editor de rotinas do RStudio e dentro de um arquivo salvo. O código não funcionará a partir do *prompt*.



Outro truque bastante útil para definir diretórios de trabalho no R é usar o símbolo `~`. Esse define a pasta 'Documentos' no *Windows*, a qual é única para cada usuário. Portanto, ao executar `setwd('~')`, irá direcionar o R a uma pasta de fácil acesso e livre modificação pelo usuário atual do computador.

## 2.17 Comentários no Código

Comentários são definidos usando o símbolo `#`. Qualquer texto a direita desse símbolo não será processado pelo R. Note que até a cor do código a direita do *hashtag* muda no RStudio. Isso dá liberdade para escrever o que for necessário dentro do *script*. Um exemplo:

```
# This is a comment
# This is another comment
x <- 'abc' # this is another comment, but mixed with code

my_l <- list(var1 = 1:10, # set var 1
            var2 = 2:5) # another var
```

**Os comentários são uma eficiente maneira de comunicar qualquer informação importante que não pode ser inferida diretamente do código.** O uso correto de comentários é tão importante quanto o código em si. Quando bem feitos, aumentam a reproducibilidade, organização e entendimento do código. Em geral, você deve evitar o uso de comentários que são muito óbvios ou muito genéricos. Por exemplo:

```
# read a csv file
df <- read.csv ('MyDataFile.csv')
```

Como você pode ver, é bastante óbvio que a linha `df <- read.csv('MyDataFile.csv')` está lendo um arquivo `.csv`. O nome da função, `read.csv` já afirma isso. Então, o comentário não foi bom pois não adicionou novas informações ao usuário. Uma melhor abordagem seria definir o autor, a descrição da funcionalidade do *script* e explicar melhor a origem e a última atualização do arquivo de dados. Vamos dar uma olhada:

```
# Script for reproducing results of JOHN (2018)
# Author: Mr Researcher (dontspamme@emailprovider.com)
# Last script update: 2018-01-10
#
# File downloaded from www.sitewithdatafiles.com/data-files/
# The description of the data goes here
#
```

```
# Last file update: 2017-12-05
df <- read.csv('MyDataFile.csv')
```

Com esses comentários, o usuário saberá o propósito do *script*, quem o escreveu e a data da última edição. A origem do arquivo e a data de atualização mais recente também estão disponíveis. Se o usuário quiser atualizar os dados, tudo o que ele tem a fazer é ir ao mencionado site e baixar o novo arquivo. Isso facilitará o uso futuro e o compartilhamento do *script*.

Outro uso de comentários é **definir seções no código**, como em:

```
# Script for reproducing results of JOHN (2018)
# Author: Mr Researcher (dontspamme@emailprovider.com)
# Last script update: 2018-01-10
#
# File downloaded from www.sitewithdatafiles.com/data-files/
# The description of the data goes here
#
# Last file update: 2017-12-05

# Clean data -----
# - remove outliers
# - remove unnecessary columns

# Create descriptive tables -----

# Estimate models -----

# Report results -----
```

O uso de uma longa linha de traços (-) é intencional. Isto faz com que o RStudio identifique as seções do código e apresente no espaço abaixo do editor de rotinas um atalho para acessar as correspondentes linhas de cada seção. Teste você mesmo, copie e cole o código acima em um novo script do RStudio, salve o mesmo, e verá que as seções aparecem em um botão entre o editor e o *prompt*. Desta forma, uma vez que você precisa mudar uma parte específica do código, você pode se dirigir rapidamente a seção desejada.



Quando começar a compartilhar código com outras pessoas, logo perceberás que os comentários são essenciais e esperados. Eles ajudam a transmitir informações que não estão disponíveis no código. Uma nota aqui, ao longo do livro você verá que os comentários do código são, na maior parte do tempo, bastante óbvios. Isso foi intencional, pois mensagens claras e diretas são importantes para novos usuários, os quais fazem parte da audiência.

## 2.18 Cancelando a Execução de um Código

Toda vez que o R estiver executando algum código, uma sinalização visual no formato de um pequeno círculo vermelho no canto direito do *prompt* irá aparecer. Caso conseguir ler (o símbolo é pequeno em monitores modernos), o texto indica o termo *stop*. Esse símbolo não somente indica que o programa ainda está rodando mas também pode ser utilizado para cancelar a execução de um código. Para isso, basta clicar no referido botão. Outra maneira de cancelar uma execução é apontar o mouse no *prompt* e pressionar a tecla *Esc* no teclado.

Para testar o cancelamento de código, copie e cole o código a seguir em um *script* do RStudio. Após salvar, rode o mesmo com `control+shift+s`.

```
for (i in 1:100) {  
  message('\nRunning code (please make it stop by hitting esc!)\n')  
  Sys.sleep(1)  
}
```

O código anterior usa um comando especial do tipo `for` para mostrar a mensagem a cada segundo. Neste caso, o código demorará 100 segundos para rodar. Caso não desejes esperar, aperte `esc` para cancelar a execução. Por enquanto, não se preocupe com as funções utilizadas no exemplo. Iremos discutir o uso do comando `for` no capítulo 8.

## 2.19 Procurando Ajuda

Uma tarefa muito comum no uso do R é procurar ajuda. A quantidade de funções disponíveis para o R é gigantesca e memorizar todas peculiaridades é quase impossível. Assim, até mesmo usuários avançados comumente procuram ajuda sobre tarefas específicas no programa, seja para entender detalhes sobre algumas funções ou estudar um novo procedimento. Portanto, saibas que o uso do sistema de ajuda do R faz parte do cotidiano.

É possível buscar ajuda utilizando tanto o painel de *help* do RStudio como diretamente do *prompt*. Para isso, basta digitar o ponto de interrogação junto ao objeto

sobre o qual se deseja ajuda, tal como em `?mean`. Nesse caso, o objeto `mean` é uma função e o uso do comando irá abrir o painel de ajuda sobre ela.

No R, toda tela de ajuda de uma função é igual, conforme se vê na Figura 2.7 apresentada a seguir. Esta mostra uma descrição da função `mean`, seus argumentos de entrada explicados e também o seu objeto de saída. A tela de ajuda segue com referências e sugestões para outras funções relacionadas. Mais importante, os **exemplos de uso da função** aparecem por último e podem ser copiados e colados para acelerar o aprendizado no uso da função.

```
mean {base} R Documentation
```

### Arithmetic Mean

**Description**

Generic function for the (trimmed) arithmetic mean.

**Usage**

```
mean(x, ...)
```

```
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

**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.

Figura 2.7: Tela de ajuda da função `mean`

Caso quiséssemos procurar um termo nos arquivos de ajuda, bastaria utilizar o comando `??"standard deviation"`. Essa operação irá procurar a ocorrência do termo em todos os pacotes do R e é muito útil para aprender como realizar alguma operação, nesse caso o cálculo de desvio padrão.

Como sugestão, o ponto inicial e mais direto para aprender uma nova função é observando o seu exemplo de uso, localizada no final da página de ajuda. Com isto, podes verificar quais tipos de objetos de entrada a mesma aceita e qual o formato e o tipo de objeto na sua saída. Após isso, leia atentamente a tela de ajuda para entender se a mesma faz exatamente o que esperas e quais são as suas opções de uso nas respectivas entradas. Caso a função realizar o procedimento desejado, podes copiar e colar o exemplo de uso para o teu próprio *script*, ajustando onde for necessário.

Outra fonte muito importante de ajuda é a própria internet. Sites como *stackoverflow.com* e *mailing lists* específicos do R, cujo conteúdo também está na internet, são fontes preciosas de informação. Havendo alguma dúvida que não foi possível solucionar via leitura dos arquivos de ajuda do R, vale o esforço de procurar uma solução via mecanismo de busca na internet. Em muitas situações, o seu problema, por mais específico que seja, já ocorreu e já foi solucionado por outros usuários.

Caso estiver recebendo uma mensagem de erro enigmática, outra dica é copiar e colar a mesma para uma pesquisa no Google. Aqui apresenta-se outro benefício do uso da língua inglesa. É mais provável que encontres a solução se o erro for escrito em inglês, dado o maior número de usuários na comunidade global. Caso não encontrar uma solução desta forma, podes inserir uma pergunta no stackoverflow ou no grupo Brasileiro do R no Facebook.



Toda vez que for pedir ajuda na internet, procure sempre 1) descrever claramente o seu problema e 2) adicionar um código reproduzível do seu problema. Assim, o leitor pode facilmente verificar o que está acontecendo ao rodar o exemplo no seu computador. Não tenho dúvida que, se respeitar ambas regras, logo uma pessoa caridosa lhe ajudará com o seu problema.

## 2.20 Utilizando *Code Completion* com a Tecla *tab*

Um dos recursos mais úteis do RStudio é o preenchimento automático de código (*code completion*). Essa é uma ferramenta de edição que facilita o encontro de nomes de objetos, nome de pacotes, nome de arquivos e nomes de entradas em funções. O seu uso é muito simples. Após digitar um texto qualquer, basta apertar a tecla *tab* e uma série de opções aparecerá. Veja a Figura 2.8 apresentada a seguir, em que, após digitar a letra *f* e apertar *tab*, aparece uma janela com uma lista de objetos que iniciam com a respectiva letra.

Essa ferramenta também funciona para pacotes. Para verificar, digite `library(r)` no *prompt* ou no editor, coloque o cursor entre os parênteses e aperte *tab*. O resultado deve ser algo parecido com a figura 2.9.

Observe que uma descrição do pacote ou objeto também é oferecida. Isso facilita bastante o dia a dia, pois a memorização das funcionalidades e dos nomes dos pacotes e os objetos do R não é uma tarefa fácil. O uso do *tab* diminui o tempo de investigação dos nomes e evita possíveis erros de digitação na definição destes.

O uso dessa ferramenta torna-se ainda mais benéfico quando os objetos são nomeados com algum tipo de padrão. No restante do livro observarás que os objetos tendem a ser nomeados com o prefixo *my*, como em `my_x`, `my_num`. O uso desse padrão facilita

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
```

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

```
Natural language support but running in an English locale
```

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

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

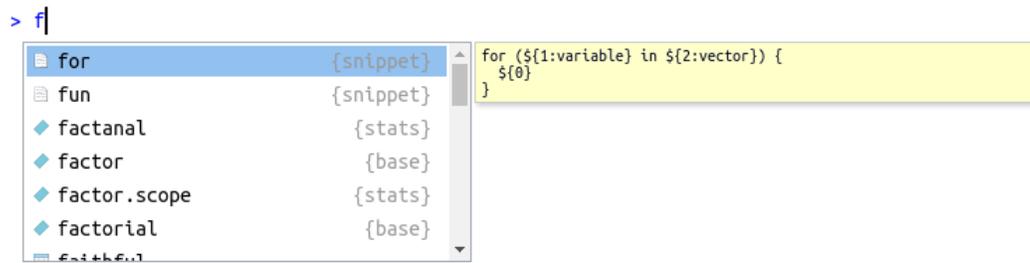


Figura 2.8: Uso do autocomplete para objetos

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
```

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

```
Natural language support but running in an English locale
```

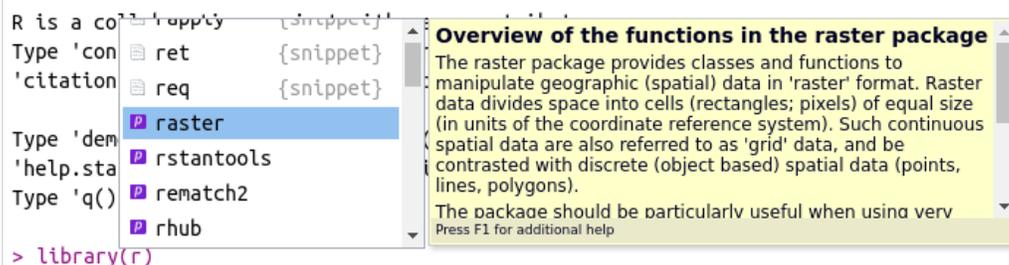


Figura 2.9: Uso do autocomplete para pacotes

o encontro futuro do nome dos objetos, pois basta digitar *my*, apertar *tab* e uma lista de todos os objetos criados pelo usuário aparecerá.

Outro uso do *tab* é no encontro de arquivos e pastas no computador. Basta criar uma variável como `my_file <- " "`, apontar o cursor para o meio das aspas e apertar a tecla *tab*. Uma tela com os arquivos e pastas do diretório atual de trabalho aparecerá, conforme mostrado na figura 2.10. Nesse caso específico, o R estava direcionado para a minha pasta de códigos, em que é possível enxergar diversos trabalhos realizados no passado.

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
```

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

```
Natural language support but running in an English locale
```

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

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

```
> my_file <- ''
```

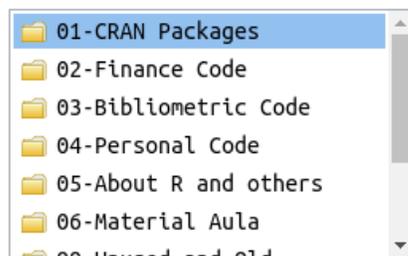


Figura 2.10: Uso do autocomplete para arquivos

Uma dica aqui é utilizar o *tab* com a raiz do computador. Assumindo que o disco do seu computador está alocado para `C:/`, digite `my_file <- "C:/"` e pressione *tab* após o símbolo `/`. Uma tela com os arquivos da raiz do computador aparecerá no RStudio. Podes facilmente navegar o sistema de arquivos utilizando as setas e *enter*.

O *autocomplete* também funciona para encontrar e definir as entradas de uma função.

Por se tratar de um tópico mais avançado, deixamos o seu uso e demonstração para o capítulo 8.



O *autocomplete* é uma das ferramentas mais importantes do RStudio, funcionando para encontro de objetos, locais no disco rígido, pacotes e funções. Acostume-se a utilizar a tecla *tab* o quanto antes e logo verá como fica mais fácil escrever código rapidamente, e sem erros de digitação.

## 2.21 Interagindo com Arquivos e o Sistema Operacional

Em muitas situações de uso do R será necessário interagir com os arquivos do computador, seja criando novas pastas, descompactando e compactando arquivos, listando e removendo arquivos do disco rígido do computador ou qualquer outro tipo de operação. Na grande maioria dos casos, o interesse é na manipulação de arquivos contendo dados.

### 2.21.1 Listando Arquivos e Pastas

Para listar arquivos do computador, basta utilizar o função `list.files`. O argumento `path` define o diretório para listar os arquivos. Na construção deste livro foi criado um diretório chamado `00-text-resources/data`, onde alguns dados são salvos. Pode-se verificar os arquivos nessa pasta com o seguinte código:

```
my_f <- list.files(path = "00-text-resources/data", full.names = TRUE)
print(my_f[1:5])
```

```
R> [1] "00-text-resources/data/AdjustedPrices-InternacionalIndices.RDATA"
R> [2] "00-text-resources/data/BovStocks_2011-12-01_2016-11-29.csv"
R> [3] "00-text-resources/data/BovStocks_2011-12-01_2016-11-29.RData"
R> [4] "00-text-resources/data/example_gethfdata.RDATA"
R> [5] "00-text-resources/data/FileWithLatinChar_ISO-8859-9.txt"
```

Observe que nesse diretório encontram-se vários arquivos `.csv`, `.rds` e `.xlsx`. Esses contêm dados que serão utilizados em capítulos futuros. Recomenda-se utilizar o argumento `full.names` como `TRUE`, o qual faz com que o retorno da função `list.files` contenha o caminho completo do arquivo. Isso facilita, por exemplo, uma possível importação de dados em que é necessário indicar não somente o nome do arquivo, mas a sua localização completa no computador. Destaca-se que também é possível listar os arquivos de forma recursiva, isto é, listar os arquivos de subpastas do endereço original. Para verificar, tente utilizar o seguinte código no seu computador:

```
# list all files recursively
list.files(path = getwd(), recursive = T, full.names = TRUE)
```

O comando anterior irá listar todos os arquivos existentes na pasta atual e subpastas de trabalho. Dependendo de onde o comando foi executado, pode levar um certo tempo para o término do processo. Caso precisar cancelar a execução, aperte *esc* no teclado.

Para listar pastas (diretórios) do computador, basta utilizar o comando `list.dirs`. Veja a seguir.

```
# list directories
my_dirs <- list.dirs(recursive = F)
print(my_dirs)
```

```
R> [1] "./_book"
R> [2] "./_bookdown_files"
R> [3] "./.Rproj.user"
R> [4] "./00-code-resources"
R> [5] "./00-text-resources"
R> [6] "./adfeR_pt_ed03_cache"
R> [7] "./adfeR_pt_ed03_files"
R> [8] "./adfeR_pt_ed03-ONLINE_cache"
R> [9] "./adfeR_pt_ed03-ONLINE_files"
R> [10] "./gdfpd2_cache"
R> [11] "./gfred_cache"
R> [12] "./many_datafiles"
R> [13] "./quandl_cache"
R> [14] "./tabs"
R> [15] "./TD Files"
```

No caso anterior, o comando lista todos os diretórios do trabalho atual sem recursividade. A saída do comando mostra os diretórios que utilizei para escrever este livro. Isso inclui o diretório de saída do livro (`./_book`), entre diversos outros. Nesse mesmo diretório, encontram-se os capítulos do livro, organizados por arquivos e baseados na linguagem *RMarkdown* (`.Rmd`). Para listar somente os arquivos com extensão `.Rmd`, utiliza-se o argumento `pattern` da função `list.files`, como a seguir:

```
list.files(path = getwd(), pattern = "*.Rmd$")
```

```
R> [1] "_BemVindo.Rmd"
R> [2] "00a-Sobre-NovaEdicao.Rmd"
R> [3] "00b-Prefacio.Rmd"
R> [4] "01-Introducao.Rmd"
```

```

R> [5] "02-Operacoes-Basicas.Rmd"
R> [6] "03-Scripts-Pesquisa.Rmd"
R> [7] "04-Importacao-Exportacao-Local.Rmd"
R> [8] "05-Importacao-Internet.Rmd"
R> [9] "06-Objetos-Armazenamento.Rmd"
R> [10] "07-Objetos-Basicos.Rmd"
R> [11] "08-Programacao-com-R--ONLINE.Rmd"
R> [12] "09-Limpando-Estruturando-Dados--ONLINE.Rmd"
R> [13] "10-Figuras--ONLINE.Rmd"
R> [14] "11-Modelagem--ONLINE.Rmd"
R> [15] "12-Reportando-resultados--ONLINE.Rmd"
R> [16] "13-Otimizacao-código--ONLINE.Rmd"
R> [17] "14-Referencias.Rmd"
R> [18] "adfeR_pt_ed03-ONLINE.Rmd"
R> [19] "index.Rmd"

```

O texto `*.Rmd$` orienta o R a procurar todos arquivos que terminam o seu nome com o texto `.Rmd`. Os símbolos `'*'` e `'$'` são operadores específicos para o encontro de padrões em texto em uma linguagem chamada *regex* (*regular expressions*) e, nesse caso, indicam que o usuário quer encontrar todos arquivos com extensão `.Rmd`. O símbolo `'*'` diz para ignorar qualquer texto anterior a `.Rmd` e `'$'` indica o fim do nome do arquivo. Os arquivos apresentados anteriormente contêm todo o conteúdo deste livro, incluindo este próprio parágrafo, localizado no arquivo `02-OperacoesBasicas.Rmd`!

### 2.21.2 Apagando Arquivos e Diretórios

A remoção de arquivos é realizada através do comando `file.remove`:

```

# create temporary file
my_file <- 'MyTemp.csv'
write.csv(x = data.frame(x=1:10),
          file = my_file)

# delete it
file.remove(my_file)

```

```
R> [1] TRUE
```

Lembre-se que debes ter permissão do seu sistema operacional para apagar um arquivo. Para o nosso caso, o retorno `TRUE` mostra que a operação teve sucesso.

Para deletar diretórios e todos os seus elementos, utilizamos `unlink`:

```
# create temp dir
dir.create('temp')

# fill it with file
my_file <- 'temp/tempfile.csv'
write.csv(x = data.frame(x=1:10),
          file = my_file)

unlink(x = 'temp', recursive = TRUE)
```

A função, neste caso, não retorna nada. Podes checar se o diretório existe com `dir.exists`:

```
dir.exists('temp')
```

```
R> [1] FALSE
```



Não preciso nem dizer, **tenha muito cuidado** com comandos `file.remove` e `unlink`, principalmente quando utilizar a recursividade (`recursive = TRUE`). Uma execução errada e partes importantes do seu disco rígido podem ser apagadas, deixando o seu computador inoperável. Vale salientar que o R **realmente apaga** os arquivos e não somente manda para a lixeira. Portanto, ao apagar diretórios com `unlink`, não poderás recuperar os arquivos.

### 2.21.3 Utilizando Arquivos e Diretórios Temporários

Um aspecto interessante do R é que ele possui uma pasta temporária que é criado na inicialização do programa. Esse diretório serve para guardar quaisquer arquivos descartáveis gerados pelo R. A cada nova sessão do R, um novo diretório temporário é criado. Ao inicializarmos o computador, essa pasta temporária é deletada.

O endereço do diretório temporário de uma sessão do R é verificado com `tempdir`:

```
my_tempdir <- tempdir()
message(str_glue('My tempdir is {my_tempdir}'))
```

```
R> My tempdir is /tmp/RtmpN2n71F
```

O último texto do diretório, neste caso `RtmpN2n71F` é aleatoriamente definido e irá trocar a cada nova sessão do R.

A mesma dinâmica é encontrada para nomes de arquivos. Caso queira, por algum motivo, utilizar um nome temporário e aleatório para algum arquivo com extensão `.txt`, utilize `tempfile` e defina a entrada `fileext`:

```
my_tempfile <- tempfile(fileext = '.txt')
message(my_tempfile)
```

```
R> /tmp/RtmpN2n71F/file11cf6322e075.txt
```

Note que o nome do arquivo – file11cf6322e075.txt – é totalmente aleatório e mudará a cada chamada de `tempfile`.

#### 2.21.4 Baixando Arquivos da Internet

O R pode baixar arquivos da Internet diretamente no código. Isso é realizado com a função `download.file`. Veja o exemplo a seguir, onde baixamos uma planilha de Excel do site da Microsoft para um arquivo temporário:

```
# set link
link_dl <- 'go.microsoft.com/fwlink/?LinkID=521962'
local_file <- tempfile(fileext = '.xlsx') # name of local file

download.file(url = link_dl,
              destfile = local_file)
```

O uso de `download.file` é bastante prático quando se está trabalhando com dados da Internet que são constantemente atualizados. Basta baixar e atualizar o arquivo com dados no início do *script*. Poderíamos continuar a rotina lendo o arquivo baixado e realizando a nossa análise dos dados disponíveis.

Um exemplo nesse caso é a tabela de empresas listadas na bolsa divulgada pela CVM (comissão de valores mobiliários). A tabela está disponível em um arquivo no site. Podemos baixar o arquivo e, logo em seguida, ler os dados.

```
library(readr)
library(dplyr)

# set destination link and file
my_link <- 'http://dados.cvm.gov.br/dados/CIA_ABERTA/CAD/DADOS/cad_cia_aberta.csv'
my_destfile <- tempfile(fileext = '.csv')

# download file
download.file(my_link,
              destfile = my_destfile,
              mode = "wb")

# read it
df_cvm <- read_csv2(my_destfile,
                    #delim = '\t',
```

```
locale = locale(encoding = 'Latin1'),
col_types = cols())
```

R> i Using "','" as decimal and "'.'" as grouping mark. Use `read\_delim()` for more cont

```
# check available columns
print(names(df_cvm))
```

```
R> [1] "CNPJ_CIA"           "DENOM_SOCIAL"
R> [3] "DENOM_COMERC"      "DT_REG"
R> [5] "DT_CONST"          "DT_CANCEL"
R> [7] "MOTIVO_CANCEL"     "SIT"
R> [9] "DT_INI_SIT"        "CD_CVM"
R> [11] "SETOR_ATIV"        "TP_MERC"
R> [13] "CATEG_REG"         "DT_INI_CATEG"
R> [15] "SIT_EMISSOR"       "DT_INI_SIT_EMISSOR"
R> [17] "CONTROLE_ACIONARIO" "TP_ENDER"
R> [19] "LOGRADOURO"        "COMPL"
R> [21] "BAIRRO"            "MUN"
R> [23] "UF"                 "PAIS"
R> [25] "CEP"                "DDD_TEL"
R> [27] "TEL"                 "DDD_FAX"
R> [29] "FAX"                 "EMAIL"
R> [31] "TP_RESP"            "RESP"
R> [33] "DT_INI_RESP"        "LOGRADOURO_RESP"
R> [35] "COMPL_RESP"         "BAIRRO_RESP"
R> [37] "MUN_RESP"           "UF_RESP"
R> [39] "PAIS_RESP"          "CEP_RESP"
R> [41] "DDD_TEL_RESP"       "TEL_RESP"
R> [43] "DDD_FAX_RESP"       "FAX_RESP"
R> [45] "EMAIL_RESP"         "CNPJ_AUDITOR"
R> [47] "AUDITOR"
```

Existem diversas informações interessantes nestes dados incluindo nome e CNPJ de empresas listadas (ou deslistadas) da bolsa de valores Brasileira. E, mais importante, o arquivo está sempre atualizado. O código anterior estará sempre buscando os dados mais recentes a cada execução.

## 2.22 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Crie um novo *script*, salve o mesmo em uma pasta pessoal. Agora, escreva os comandos no script que definam dois objetos: um contendo uma sequência entre 1

e 100 e outro com o texto do seu nome (ex. 'Ricardo'). Execute o código com os atalhos no teclado.

02 - No *script* criado anteriormente, use função `message` para mostrar a seguinte frase no *prompt* do R: "My name is ....".

03 - Dentro do mesmo script, mostre o diretório atual de trabalho (veja função `getwd`, tal como em `print(getwd())`). Agora, modifique o seu diretório de trabalho para o *Desktop* (*Área de Trabalho*) e mostre a seguinte mensagem na tela do *prompt*: 'My desktop address is ....'. Dica: use e abuse da ferramenta *autocomplete* do RStudio para rapidamente encontrar a pasta do *desktop*.

04 - Utilize o R para baixar o arquivo compactado com o material do livro, disponível nesse link<sup>3</sup>. Salve o mesmo como um arquivo na pasta temporária da sessão (veja função `tempfile`).

05 - Utilize a função `unzip` para descompactar o arquivo baixado na questão anterior para um diretório chamado 'adfeR-Files' dentro da pasta do "Desktop". Quantos arquivos estão disponíveis na pasta resultante? Dica: use o argumento `recursive = TRUE` com `list.files` para procurar também todos subdiretórios disponíveis.

06 - Toda vez que o usuário instala um pacote do R, os arquivos particulares ao pacote são armazenados localmente em uma pasta específica do computador. Utilizando comando `Sys.getenv('R_LIBS_USER')` e `list.dirs`, liste todos os diretórios desta pasta. Quantos pacotes estão disponíveis nesta pasta do seu computador?

07 - No mesmo assunto do exercício anterior, liste todos os arquivos em todas as subpastas do diretório contendo os arquivos dos diferentes pacotes. Em média, quantos arquivos são necessários para cada pacote?

08 - Use função `install.packages` para instalar o pacote `BatchGetSymbols` no seu computador. Após a instalação, use função `BatchGetSymbols::BatchGetSymbols` para baixar dados de preços para a ação da Petrobrás – PETR3 (PETR3.SA no Yahoo finance) – nos últimos 15 dias. Dicas: 1) use função `Sys.Date()` para definir data atual e `Sys.Date() - 15` para calcular a data localizada 15 dias no passado; 2) note que a saída de `BatchGetSymbols` é uma lista, um tipo especial de objeto, e o que os dados de preços estão localizados no segundo elemento dessa lista.

09 - O pacote `cranlogs` permite o acesso a estatísticas de *downloads* de pacotes do CRAN. Após instalar o `cranlogs` no seu computador, use função `cranlogs::cran_top_downloads` para verificar quais são os 10 pacotes mais instalados pela comunidade global no último mês. Qual o pacote em primeiro lugar? Dica: Defina a entrada da função `cran_top_downloads` como sendo `when = 'last-month'`. Também note que a resposta aqui pode não ser a mesma que obteve pois esta depende do dia em que foi executado o código.

---

<sup>3</sup><https://www.msperlin.com/blog/files/adfer-files/adfeR-code-and-data.zip>

- a) dplyr
- b) vctrs
- c) ggplot2
- d) cli
- e) ragg

10 - Utilizando pacote `devtools`, instale a versão de desenvolvimento do pacote `ggplot2`, disponível no repositório de Hadley Wickham. Carregue o pacote usando `library` e crie uma figura simples com o código `qplot(y = rnorm(10), x = 1:10)`.

11 - Utilizando sua capacidade de programação, verifique no seu computador qual pasta, a partir do diretório de “Documentos” (atalho = `~`), possui o maior número de arquivos. Apresente na tela do R as cinco pastas com maior número de arquivos.

# Capítulo 3

## Desenvolvendo Rotinas de Pesquisa

No capítulo anterior aprendemos a utilizar o R e RStudio para operações básicas tal como a criação de *scripts*, manipulação de objetos no R, mudança de diretório de trabalho, entre outras. Antes de passarmos para a importação de dados, é necessário discutirmos como organizar uma rotina de pesquisa no seu computador.

Neste capítulo iremos tratar das etapas de pesquisa e a organização de arquivos no computador, incluindo dados e rotinas. O principal objetivo aqui é mostrar e justificar um formato de trabalho que facilite o desenvolvimento e compartilhamento de código. Por exemplo, ao abrir um diretório de projeto antigo, a própria estrutura dos arquivos e pastas já indicará como o mesmo funciona e quais as suas entradas e saídas.

### 3.1 Etapas de uma Pesquisa

Um *script* de pesquisa pode ser organizado em quatro etapas consecutivas:

1. **Importação dos dados:** Dados crus (originais) do mundo real são importados para a sua sessão do R, seja de arquivo local ou da internet. Neste momento, nenhum tipo de manipulação de dados ou reformatação manual deve acontecer. Como regra de bolso, **dados crus nunca devem ser tocados**. Deixe toda e qualquer manipulação para a etapa seguinte.
2. **Limpeza e estruturação dos dados:** Dados importados na fase anterior são processados em uma etapa de limpeza e estruturação. Registros anormais e erros em observações são eliminados ou tratados das tabelas originais. Novas variáveis de interesse são criadas com base nos dados importados. A estrutura dos dados originais também pode ser refeita de acordo com a necessidade. O

resultado final dessa etapa deve ser, preferencialmente, uma tabela final com todos os dados necessários para a análise do problema.

3. **Visualização de dados e teste de hipóteses:** Após limpar e estruturar os dados, o próximo passo é realizar a análise em si, a qual será visual, através da criação de diversas figuras, ou então com a estimação de modelos e testes de hipótese. Essa etapa refere-se ao próprio problema da pesquisa. Na grande maioria dos casos, essa é a fase que exigirá maior trabalho, visto que representa o coração da pesquisa. Essa etapa termina com a criação de arquivos, tal como uma figura com extensão *.png*, que serão utilizados na etapa seguinte.
4. **Reportando os resultados:** A última etapa é a criação dos objetos de interesse a serem reportados no artigo ou relatório. Esses objetos geralmente referem-se a tabelas e figuras, que podem ser exportados como arquivos externos e futuramente importados em um relatório.

Cada uma das etapas anteriores pode ser estruturada em um único arquivo *.R* ou em vários arquivos separados. O uso de vários arquivos é preferível quando as etapas de cada pesquisa demandarem muito tempo de processamento. Por exemplo, na importação e organização de uma base de dados de grande volume, vale a pena separar os procedimentos em arquivos diferentes. Isto facilita o encontro de erros e manutenção do código.

Um caso prático seria a análise de dados volumosos de transações financeiras no mercado de capitais. A importação e limpeza desses dados leva muito tempo. Uma organização inteligente da estrutura da pesquisa seria dividir as etapas em diferentes *scripts* do R e, usando arquivos locais de dados como “pontes”, lincar um script com outro. Assim, a rotina de importação e limpeza de dados salva um arquivo *.rds* no final de sua execução, o qual é importando no *script* de teste de hipóteses. Desta forma, uma mudança na modelagem do problema não exige que todo o processo de limpeza dos dados seja refeito. Essa simples organização de arquivos economiza bastante tempo. A lógica é clara: isole as partes da pesquisa que exigem muito tempo de execução – e pouco de desenvolvimento –, e conecte as mesmas com o resto do código usando arquivos externos de dados, tal como arquivos *.rds* e *.csv*.

Caso você for trabalhar com diversos arquivos, uma sugestão é criar uma estrutura de nomenclatura que defina as etapas da pesquisa. Um exemplo seria nomear o código de importação de dados como `01-Import-and-clean-data.R`, o código de modelagem como `02-build-report-models.R` e assim sucessivamente. O efeito prático é que o uso de um contador na primeira letra do nome do arquivo faz com que a ordem de execução do programa fique clara. Indo além, você pode criar um *script* mestre chamado `00-run-it-all.R` que roda (comando `source`) todos as outras rotinas. Assim, toda vez que realizar uma atualização nos dados originais, você pode simplesmente executar `00-run-it-all.R` e terá os novos resultados, sem necessitar rodar cada *script* individualmente.

## 3.2 A Estrutura de Diretórios

Uma estrutura de organização de diretórios também beneficia a reprodutibilidade e organização da pesquisa. Para rotinas simples, com uma base de dados única e um baixo número de procedimentos, não é necessário dispendir muito tempo pensando na organização dos arquivos nos diretórios. Para programas mais complexos, onde existem diversas etapas de pesquisa e diversas bases de dados, uma segmentação dos diretórios é não somente recomendada como essencial.

Uma estrutura de diretórios que considero eficiente é criar um diretório único para a pesquisa e, dentro dele, criar subdiretórios para cada elemento de entrada e saída. Por exemplo, você pode criar um subdiretório chamado `data`, onde todos os dados originais serão guardados, um diretório `figs` e um `tables`, para onde figuras e tabelas com resultados de interesse serão exportadas. Para funções utilizadas na pesquisa, você pode também criar um diretório chamado `R-Fcts`. Todos os *scripts* principais da pesquisa, e nada mais, devem ficar na raiz do diretório de pesquisa. Um exemplo da estrutura de arquivos seria:

```
/Capital Markets and Inflation/
  /data/
    stock_indices.csv
    inflation_data.csv
  /figs/
    SP500_and_inflation.png
  /tables/
    Table1_descriptive_table.tex
    Table2_model_results.tex
  /R-Fcts/
    fct_models.R
    fct_clean_data.R
  0-run-it-all.R
  1-import-and-clean-data.R
  2-run-research.R
```

O código de pesquisa também deve ser independente, com todos os arquivos disponíveis em uma subpasta do diretório raiz. Se você estiver usando muitos pacotes diferentes, é aconselhável adicionar um comentário nas primeiras linhas de `0-run-it-all.R` que indica quais pacotes são necessários para executar o código. A forma mais amigável de informar é adicionando uma linha comentada que instala todos os pacotes necessários, como em `# install.packages ('pkg1', 'pkg2', ...)`. Portanto, quando alguém recebe o código pela primeira vez, tudo o que ele (ou ela) precisa fazer é retirar o símbolo de comentário e executar a rotina. Dependências externas e etapas para a instalação correta do software também devem ser informadas.

Os benefícios deste formato de diretório são os seguintes. Se você precisar compartilhar o código com outros pesquisadores, basta compactar o diretório em um único arquivo e enviá-lo ao destinatário. Após descompactar o arquivo, a estrutura da pasta informa imediatamente ao usuário onde deve alterar os dados originais, a ordem de execução dos scripts na pasta raiz e onde as saídas são salvas. O mesmo benefício acontece quando você reutiliza seu código no futuro, digamos, daqui a três anos. Ao trabalhar de forma mais inteligente, você será mais produtivo, gastando menos tempo com etapas repetitivas e desnecessárias.

Seguindo a sugestão de um *script* mestre, um exemplo comentado para o conteúdo do arquivo `00-run-it-all.R` seria:

```
# clean up workspace
rm(list=ls())

# close all figure windows created with x11()
graphics.off()

# load packages
library(pkg1)
library(pkg2)
library(pkg3)

# change directory
my_dir <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(my.d)

# list functions in 'R-Fcts'
my_R_files <- list.files(path='R-Fcts',
                        pattern = '*.R',
                        full.names=TRUE)

# Load all functions in R
sapply(my_R_files,source)

# Import data script
source('01-import-and-clean-data.R')

# run models and report results
source('02-run-research.R')
```

Essa é a primeira vez que usamos as funções `graphics.off` e `sapply`. A primeira fecha todas as janelas de gráficos abertas. Essas tendem a acumular no decorrer do trabalho e devem ser fechadas no início de um novo *script*. O comando `sapply` aplica

uma função, nesse caso `source`, para uma série de elementos. O efeito prático em `sapply(my_R_files, source)` é que todos arquivos com extensão `.R` localizados na pasta `R-Fct` serão executados. Ou seja, todas funções que escrevermos nos arquivos `fct_models.R` e `fct_clean_data.R` serão carregadas em nossa sessão de trabalho. Futuramente, capítulos 10 e 8, iremos voltar ao assunto de uso de funções customizadas.

Note que, assumindo que todos os pacotes já estão instalados no computador, o script `00-run-it-all.R` é facilmente compartilhável e irá rodar em outro computador com nenhum problema. Caso o leitor quiser ir um passo além, pode também utilizar a função `file.copy` para copiar todos os arquivos de figuras para a pasta de escrita do artigo ou documento acadêmico. A partir disso, crie um link no texto para cada arquivo copiado anteriormente. Como exemplo, no LaTeX você pode incluir um arquivo de figura com o comando `\includegraphics{filenamehere}`. Pode também criar um link direto entre o arquivo de escrita e a figura da pesquisa, apesar de esse método não ser recomendado, uma vez que ele cria uma dependência externa ao arquivo de escrita. Em ambas as formas, todas as figuras da pesquisa serão automaticamente atualizadas no texto e estarão sincronizadas com os arquivos provenientes do código da pesquisa. Para tabelas, a importação não é tão simples, pois uma tabela pode ser escrita em diversos formatos. Existem, porém, pacotes específicos para lidar com isso. No capítulo 12 estudaremos uma forma eficiente de reportar resultados utilizando os pacotes `xtable` (Dahl et al., 2019), `texreg` (Leifeld, 2022), entre outros.

### 3.3 Pontos Importantes em uma Pesquisa

Aproveitando o tópico de execução de pesquisa, vou colocar aqui algumas sugestões para a realização de pesquisas com o R. Deixo claro que essas são posições pessoais, oriundas das minha experiência de trabalho. Muitos pontos levantados aqui são específicos para o ambiente acadêmico, porém podem ser facilmente estendíveis para a prática de pesquisa fora das universidades.

**Em primeiro lugar, conheça os seus dados!** Entendo que o primeiro instinto ao se deparar com uma nova base de dados é instantaneamente importá-la no R e sair realizando análises. Aqui, um certo nível de cautela é necessário. Toda vez que se deparar com um conjunto de dados novos, se pergunte o quanto você **realmente** conhece esses dados:

- Como os dados foram coletados? Para que fim?
- Como estes dados se comparam com dados já utilizados em outros trabalhos?
- Existe alguma possibilidade de viés na forma de coleta dos dados?

Lembre-se que o propósito final de qualquer pesquisa é a comunicação. Certamente irás reportar os resultados para pessoas que irão ter algum tipo de opinião informada

sobre a pesquisa. É provável que os avaliadores terão mais experiência que você no assunto, incluindo sobre as fontes e individualidades dos dados. Não desejo para ninguém estar em uma situação onde um esforço de pesquisa, com investimento de 3 a 6 meses de trabalho entre programação e escrita, é anulado por um simples lapso na checagem dos dados. Infelizmente, isso não é incomum.

Portanto, **seja muito cauteloso sobre os dados que estás utilizando**. Um detalhe que passa despercebido pode invalidar toda uma pesquisa. Caso tiver sorte e a base de dados vier acompanhada de um manual escrito, destrinche o mesmo até os últimos detalhes. Elenque as principais dúvidas em relação aos dados e, em caso das informações não estarem claras, não seja tímido em enviar os questionamentos para o responsável.

O segundo ponto é o código. Após terminar de ler este livro, o seu computador se tornará um poderoso aliado em fazer realidade as suas ideias de pesquisa, por mais gigantescas e mirabolantes que forem. Porém, **um grande poder vem acompanhado de grande responsabilidade**. Um erro de código pode facilmente inviabilizar ou tendenciar a sua pesquisa.

Lembre que analisar dados é a sua profissão e a **sua reputação é o seu maior ativo**. Caso não tenhas confiança no código produzido, não publique ou comunique os seus resultados. O código de sua pesquisa é de total responsabilidade sua e de mais ninguém. Verifique e questione o mesmo quantas vezes for necessário. Seja, sempre, o avaliador mais criterioso do seu trabalho:

- As estatísticas descritivas das variáveis relatam fielmente a base de dados?
- Existe alguma relação entre as variáveis que pode ser verificada na tabela descritiva?
- Os resultados encontrados fazem sentido para a literatura atual do assunto? Caso não, como explicá-los?
- É possível que um *bug* no código tenha produzido o resultado encontrado?

Ainda me surpreendo como pesquisas submetidas a respeitados periódicos podem ser negadas a publicação baseado em uma simples análise da tabela descritiva dos dados construídos. Erros básicos de cálculos de variáveis são facilmente encontrados para um olho treinado, que sabe onde procurar. Esse processo de avaliação contínua da sua pesquisa não somente o deixará mais forte como pesquisador(a) mas também servirá de treino para a prática de avaliação de pares, muito utilizada na pesquisa acadêmica. Caso não tenhas confiança suficiente para reportar os resultados, teste o seu código ostensivamente. Caso já o tenha feito e ainda não estás confiante, identifique as linhas de código que tens mais dificuldade e busque ajuda com um colega ou o seu orientador, caso existir. Este último é um forte aliado que pode ajudá-lo com a sua maior experiência.

Todo o trabalho de pesquisa é, de certa forma, baseado em trabalhos já existentes.

Atualmente é extremamente difícil realizar algum tipo de pesquisa que seja totalmente inovadora. O conhecimento é construído na forma de blocos, um sobre o outro. Sempre existe uma parcela de literatura que deve ser consultada. Particularmente para o caso de pesquisa em dados, debes sempre comparar os seus resultados com os resultados já apresentados na literatura do assunto, principalmente quando é um estudo replicado. Caso os resultados principais não forem semelhantes aos encontrados na literatura, questione-se o porquê disso. Será que um erro de código pode ter criado esse resultado?

Deixo claro que é possível sim que resultados de uma pesquisa sejam diferentes dos da literatura, porém, o contrário é mais provável. O conhecimento disso demanda cuidado com o seu código. *Bugs* e erros de código são bastante comuns, principalmente nas primeiras versões das rotinas. É importante reconhecer este risco e saber administrá-lo.

### 3.4 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Imagine uma pesquisa envolvendo a análise do seu orçamento doméstico ao longo do tempo. Dados estão disponíveis em planilhas eletrônicas separadas por mês, durante 10 anos. O objetivo da pesquisa é entender se é possível a compra de um imóvel daqui a cinco anos. A partir disso, detalhe em texto os elementos em cada etapa do estudo, desde importação dos dados até a construção do relatório.

02 - Com base no estudo proposto anteriormente, crie uma estrutura de diretórios em seu computador para acomodar a pesquisa. Em um arquivo texto na pasta raiz da pesquisa, assinale arquivos fictícios para cada subdiretório (veja estrutura de diretórios no início da seção 3.2). Note que a criação dos diretórios pode ser realizada pelo próprio R.



# Capítulo 4

## Importação e Exportação de Dados Locais

Sem dúvida, a primeira etapa de um *script* de pesquisa é carregar os seus dados em uma sessão do R. Neste capítulo iremos aprender a importar e exportar dados contidos em arquivos locais. Apesar de não ser uma tarefa particularmente difícil, um analista de dados deve entender as diferentes características de cada formato de arquivo e como tirar vantagem deste conhecimento em cada situação. Enquanto algumas facilitam a colaboração e troca de dados, outras podem oferecer um ganho significativo em tempo de execução na leitura e gravação.

Aqui iremos traçar uma lista abrangente com os seguintes formatos e extensões de arquivos:

- Dados delimitados em texto (*csv*);
- Microsoft Excel (*xls,xlsx*);
- Arquivos de dados nativos do R (*RData* e *rds*)
- Formato **fst** (*fst*)
- SQLite (*SQLITE*)
- Texto não estruturado (*txt*).

A primeira lição na importação de dados para o R é que o local do arquivo deve ser indicado explicitamente no código. Este endereço é passado para a função que irá ler o arquivo. Veja a definição a seguir:

```
my_file <- 'C:/Data/MyData.csv'
```

Note o uso de barras (/) para designar o diretório do arquivo. Referências relativas também funcionam, tal como em:

```
my_file <- 'Data/MyData.csv'
```

Neste caso, assume-se que na pasta atual de trabalho existe um diretório chamado `Data` e, dentro desse, um arquivo denominado `MyData.csv`. Se o endereço do arquivo é simplesmente o seu nome, assume-se que o mesmo encontra-se na raiz da pasta de trabalho. Para verificar o endereço atual de trabalho, utilize a função `getwd`.



Aqui novamente reforço o uso do *tab* e *autocomplete* do RStudio. É muito mais **fácil e prático** encontrar arquivos do disco rígido do computador usando a navegação via *tab* do que copiar e colar o endereço do seu explorador de arquivos. Para usar, abra aspas no RStudio, coloque o cursor do *mouse* entre as aspas e aperte *tab*.

Um ponto importante aqui é que os **dados serão importados e exportados no R como objetos do tipo `dataframe`**. Isto é, uma tabela contida em um arquivo Excel ou *.csv* se transformará em um objeto do tipo `dataframe` no ambiente de trabalho do R. Quando exportarmos dados, o formato mais comum é esse mesmo tipo de objeto. Convenientemente, `dataframes` são nada mais que tabelas, com linhas e colunas.

Cada coluna do `dataframe` importado terá a sua própria classe, sendo as mais comuns numérica (*numeric*), texto (*character*), fator (*factor*) e data (*Date*). Quando realizando a importação, **é de fundamental importância que os dados sejam representados na classe correta**. Uma vasta quantidade de erros podem ser evitados pela simples checagem das classes das colunas no `dataframe` resultante do processo de importação. Por enquanto somente é necessário entender esta propriedade básica de `dataframes`. Estudaremos esse objeto mais profundamente no capítulo 6.

## 4.1 Pacote `adfeR`

Nas seções futuras iremos utilizar o pacote do livro – `adfeR` – para carregar diversos exemplos de arquivos. Se você seguiu as instruções da seção *Material Suplementar* localizada no prefácio do livro, já deves ter o pacote instalado. Caso contrário, execute o seguinte código:

```
# install devtools (if not installed)
if (!require(devtools)) install.packages ('devtools')

# install book package
devtools::install_github ('msperlin/adfeR')
```

Uma vez que você instalou o pacote `adfeR`, todos os arquivos de dados usados no livro foram baixados. Podemos verificar os cinco primeiros arquivos disponíveis com o comando `adfeR::list_available_data`:

```
# list available data files
print(adfeR::list_available_data()[1:5])

R> [1] "batchgetsymbols_parallel_example.rds"
R> [2] "Brazil_footbal_games.csv"
R> [3] "example_tsv.csv"
R> [4] "FileWithLatinChar_Latin1.txt"
R> [5] "FileWithLatinChar_UTF-8.txt"
```

Os arquivos anteriores estão salvos na pasta de instalação dos pacote `adfeR`. Para ter o caminho completo, basta usar função `adfeR::get_data_file` tendo o nome do arquivo como entrada:

```
# get location of file
my_f <- adfeR::get_data_file('grunfeld.csv')

# print it
print(my_f)
```

A partir de agora iremos usar a função `adfeR::get_data_file` para obter o caminho dos arquivos utilizados nos exemplos. Note que, desde que tenha o pacote `adfeR` instalado, puedes facilmente reproduzir todos os exemplos do livro no seu computador.

## 4.2 Arquivos *csv*

Considere o arquivo de dados no formato `csv` chamado `'Ibov.csv'`, pertencente ao repositório do livro. Vamos copiar o mesmo para a pasta “Meus Documentos” com o uso do tilda (`~`):

```
# get location of file
my_f <- adfeR::get_data_file('Ibov.csv')

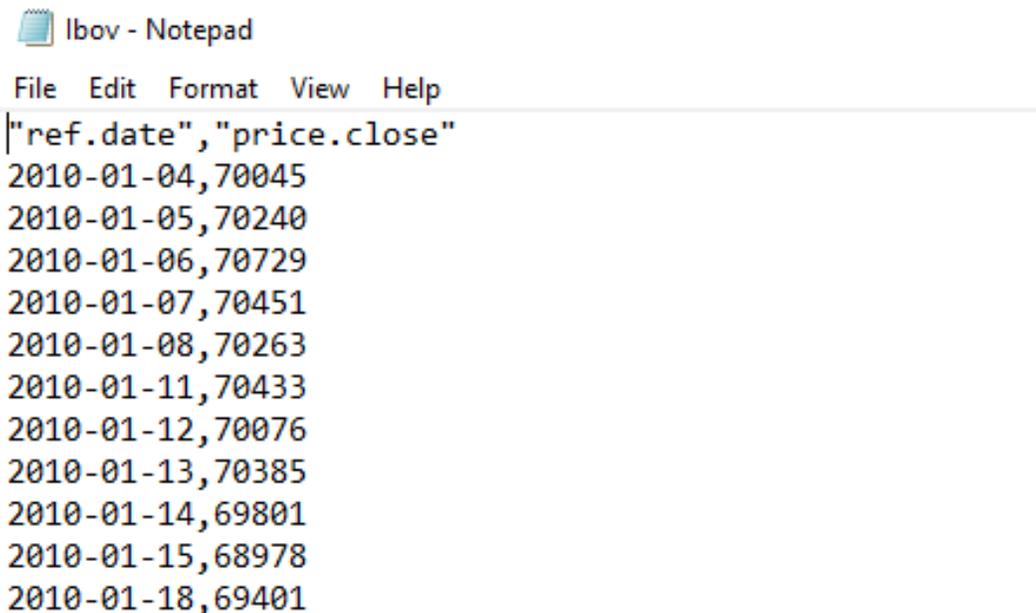
# copy to ~
file.copy(from = my_f,
          to = '~' )
```

```
R> [1] TRUE
```

```
R> [1] TRUE
```

Caso seja a primeira vez trabalhando com arquivos do tipo `.csv`, sugiro usar o ex-

plorador de arquivos do Windows e abrir Ibov.csv com qualquer editor de texto instalado, tal como o *Notepad* (veja figura 4.1). Observe que as primeiras linhas do arquivo definem os nomes das colunas: “ref.date” e “price.close”. Conforme notação internacional, as linhas são definidas pela quebra do texto e as colunas pelo uso da vírgula (,).



```
Ibov - Notepad
File Edit Format View Help
|"ref.date", "price.close"
2010-01-04,70045
2010-01-05,70240
2010-01-06,70729
2010-01-07,70451
2010-01-08,70263
2010-01-11,70433
2010-01-12,70076
2010-01-13,70385
2010-01-14,69801
2010-01-15,68978
2010-01-18,69401
```

Figura 4.1: Ibov.csv no Notepad



Quando trabalhando com dados brasileiros, a notação internacional pode gerar uma confusão desnecessária. Dados locais tendem a usar a vírgula para indicar valores decimais em números. Assim, é comum que dados locais do Brasil sejam exportados usando a semi-vírgula (;) como separador de colunas e a própria vírgula como símbolo de decimais. Como regra de bolso, **nunca mude o formato no arquivo de texto original**. Deixe esse serviço para o próprio R e seus pacotes.

O conteúdo de Ibov.csv é bastante conservador e não será difícil importar o seu conteúdo. Porém, saiba que muitas vezes o arquivo *.csv* vem com informações extras de cabeçalho – o chamado *metadata* – ou diferentes formatações que exigem adaptações. Como sugestão para evitar problemas, antes de prosseguir para a importação de dados em um arquivo *.csv*, abra o arquivo em um editor de texto qualquer e siga os seguintes passos:

- 1) Verifique a existência de texto antes dos dados e a necessidade de ignorar

algumas linhas iniciais. A maioria dos arquivos `.csv` não contém cabeçalho, porém deve sempre checar. No R, a função de leitura de arquivos `.csv` possui uma opção para ignorar um definido número de linhas antes de começar a leitura do arquivo;

- 2) Verifique a existência ou não dos nomes das colunas na primeira linha com os dados. Em caso negativo, verifique com o autor qual o nome (e significado) das colunas;
- 3) Verifique qual o símbolo de separador de colunas. Comumente, seguindo notação internacional, será a vírgula, porém nunca se tem certeza sem checar;
- 4) Para dados numéricos, verifique o símbolo de decimal, o qual deve ser o ponto (.) tal como em 2.5. Caso necessário, pode ajustar o símbolo na própria função de leitura;
- 5) Verifique a codificação do arquivo de texto. Normalmente é UTF-8, Latin1 (ISO-8859) ou windows1252. Esses são formatos amplos e devem ser suficientes para a maioria dos idiomas. Sempre que você encontrar símbolos estranhos nas colunas de texto do `dataframe` resultante, o problema é devido a uma diferença na codificação entre o arquivo e o R. Os usuários do Windows podem verificar a codificação de um arquivo de texto abrindo-o no software Notepad++<sup>1</sup>. As informações sobre a codificação estarão disponíveis no canto inferior direito do editor. No entanto, você precisa estar ciente de que o Notepad++ não faz parte da instalação do Windows e pode ser necessário instalá-lo em seu computador. Os usuários de Linux e Mac podem encontrar as mesmas informações em qualquer software editor de texto avançado, como o Kate<sup>2</sup>.



Sempre que você encontrar uma estrutura de texto inesperada em um arquivo `.csv`, use os argumentos da função de leitura `csv` para importar as informações corretamente. Repetindo, **nunca modifique dados brutos manualmente**. É muito mais eficiente usar o código R para lidar com diferentes estruturas de arquivos em `.csv`. Pode parecer mais trabalhoso, mas essa política vai economizar muito tempo no futuro, pois, em algumas semanas, você provavelmente esquecerá como limpou manualmente aquele arquivo `.csv` utilizado em pesquisa passada. Com o uso de código para a adaptação da importação de dados, sempre que você precisar atualizar o arquivo de dados, o código irá resolver todos os problemas, automatizando o processo.

---

<sup>1</sup><https://notepad-plus-plus.org/>

<sup>2</sup><https://kate-editor.org/>

### 4.2.1 Importação de Dados

O R possui uma função nativa chamada `base::read.csv` para importar dados de arquivos `.csv`. Porém, esse é um dos muitos casos em que a alternativa do `tidyverse` – `readr::read_csv` – é mais eficiente e mais fácil de trabalhar. Resumindo, `readr::read_csv` lê arquivos mais rapidamente que `base::read.csv`, além de usar regras mais inteligentes para definir as classes das colunas importadas.

Este é a primeira vez que usamos um pacote do `tidyverse`, neste caso o `readr`. Antes de fazer isso, é necessário instalá-lo em sua sessão R. Uma maneira simples de instalar todos os pacotes pertencentes ao `tidyverse` é instalar o módulo de mesmo nome:

```
install.packages('tidyverse')
```

Após executar o código anterior, todos os pacotes `tidyverse` serão instalados em seu computador. Você também deve ter em mente que alguns aspectos dessa instalação podem demorar um pouco. Assim que terminar, carregue o conjunto de pacotes `tidyverse`.

```
# load library
library(tidyverse)
```

De volta à importação de dados de arquivos `.csv`, use a função `readr::read_csv` para carregar o conteúdo do arquivo `Ibov.csv` no R:

```
# set file to read
my_f <- adfeR::get_data_file('Ibov.csv')

# read data
my_df_ibov <- read_csv(my_f)
```

```
R> Rows: 2716 Columns: 2
R> -- Column specification -----
R> Delimiter: ","
R> dbl (1): price.close
R> date (1): ref.date
R>
R> i Use `spec()` to retrieve the full column specification for this data.
R> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

O conteúdo do arquivo importado é convertido para um objeto do tipo `dataframe` no R. Conforme mencionado no capítulo anterior, cada coluna de um `dataframe` tem uma classe. Podemos verificar as classes de `my_df_ibov` usando a função `glimpse` do pacote `dplyr`, que também faz parte do `tidyverse`:

```
# check content
glimpse(my_df_ibov)
```

```
R> Rows: 2,716
R> Columns: 2
R> $ ref.date    <date> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Observe que a coluna de datas (`ref.date`) foi importada como um vetor `Date` e os preços de fechamento como numéricos (`dbl`, precisão dupla). Isso é exatamente o que esperávamos. Internamente, a função `read_csv` identifica as classes das colunas de acordo com seu conteúdo.

Observe também como o código anterior apresentou a mensagem “Parsed with column specification: ...”. Essa mensagem mostra como a função identifica as classes das colunas lendo as primeiras 1000 linhas do arquivo. Regras inteligentes tentam prever a classe com base no conteúdo importado. Podemos usar essas informações em nosso próprio código copiando o texto e atribuindo-o a uma variável:

```
# set cols from readr import message
my_cols <- cols(
  price.close = col_double(),
  ref.date = col_date(format = "")
)

# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
  col_types = my_cols)
```

Como um exercício, vamos importar os mesmos dados, porém usando a classe `character` (texto) para colunas `ref.date`:

```
# set cols from readr import message
my_cols <- cols(
  price.close = col_double(),
  ref.date = col_character()
)

# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
  col_types = my_cols)

# check content
glimpse(my_df_ibov)
```

```
R> Rows: 2,716
R> Columns: 2
R> $ ref.date      <chr> "2010-01-04", "2010-01-05", "2010-01-0~
R> $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Como esperado, a coluna de datas – `ref.date` – agora foi importada como texto. Assim, o uso de `readr::read_csv` pode ser resumido em duas etapas: 1) leia o arquivo sem argumentos em `read_csv`; 2) copie o texto das classes de coluna padrão da mensagem de saída e adicione como entrada `col_types`. O conjunto de passos anterior é suficiente para a grande maioria dos casos. O uso da mensagem com as classes das colunas é particularmente útil quando o arquivo importado tem várias colunas e a definição manual de cada classe exige muita digitação.

Uma alternativa mais prática no uso do `read_csv` é confiar na heurística da função e usar a definição padrão das colunas automaticamente. Para isto, basta definir a entrada `col_types` como `cols()`. Veja a seguir:

```
# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
                       col_types = cols())
```

Agora, vamos estudar um caso mais anormal de arquivo `.csv`. No pacote do livro temos um arquivo chamado `funky_csv_file.csv` onde:

- o cabeçalho possui texto com informações dos dados;
- o arquivo usará a vírgula como decimal;
- o texto do arquivo conterá caracteres latinos.

As primeiras 10 linhas dos arquivos contém o seguinte conteúdo:

```
R> Exemplo de arquivo .csv com formato alternativo:
R> - colunas separadas por ";"
R> - decimal como ","
R>
R> Dados retirados em 2021-01-13
R> Origem: www.funkysite.com.br
R>
R> COD;UF;COD;NOME;state;SIGLA;number_col
R> 35;3546306;Santa Cruz das Palmeiras;São Paulo; SP;1,90208656713367
R> 21;2103109;Cedral;Maranhão; MA;69,8087496915832
```

Note a existência do cabeçalho até linha de número 7 e as colunas sendo separadas pela semi-vírgula (“;”).

Ao importar os dados com opções padrões (e erradas), teremos o resultado a seguir:

```
my_f <- adfeR::get_data_file('funky_csv_file.csv')

df_funky <- read_csv(my_f,
                    col_types = cols())

glimpse(df_funky)
```

```
R> Rows: 2
R> Columns: 1
R> $ `Exemplo de arquivo .csv com formato alternativo:` <chr> ~
```

Claramente a importação deu errado, com a emissão de diversas mensagens de *warning*. Para resolver, utilizamos o seguinte código, estruturando todas as particularidades do arquivo:

```
df_not_funky <- read_delim(file = my_f,
                          skip = 7, # how many lines do skip
                          delim = ';', # column separator
                          col_types = cols(), # column types
                          locale = locale(decimal_mark = ',')# locale
)

glimpse(df_not_funky)
```

```
R> Rows: 100
R> Columns: 6
R> $ COD.UF      <dbl> 35, 21, 35, 35, 41, 31, 31, 21, 29, 26, ~
R> $ COD        <dbl> 3546306, 2103109, 3514700, 3538105, 411~
R> $ NOME       <chr> "Santa Cruz das Palmeiras", "Cedral", "~
R> $ state      <chr> "São Paulo", "Maranhão", "São Paulo", "~
R> $ SIGLA      <chr> " SP", " MA", " SP", " SP", " PR", " MG~
R> $ number_col <dbl> 1.902087, 69.808750, 81.509312, 56.8400~
```

Veja que agora os dados foram corretamente importados, com as classes corretas das colunas. Para isso, usamos a função alternativa `readr::read_delim`. O pacote `readr` também possui várias outras funções para situações específicas de importação. Caso a função `read_csv` não resolva o seu problema na leitura de algum arquivo de dados estruturado em texto, certamente outra função desse pacote resolverá.

### 4.2.2 Exportação de Dados

Para exportar tabelas em um arquivo `.csv`, basta utilizar a função `readr::write_csv`. No próximo exemplo iremos criar dados artificiais, salvar em um dataframe e exportar para um arquivo `.csv` temporário. Veja a seguir:

```

library(readr)

# set number of observations
N <- 100

# create dataframe with random data
my_df <- data.frame(y = runif(N),
                   z = rep('a', N))

# write to file
f_out <- tempfile(fileext = '.csv')
write_csv(x = my_df, file = f_out)

```

No exemplo anterior, salvamos o `dataframe` chamado `my_df` para o arquivo `file11cf2f714b38.csv`, localizado na pasta temporária do computador. Podemos verificar o arquivo importando o seu conteúdo:

```

my_df <- read_csv(f_out,
                 col_types = cols(y = col_double(),
                                 z = col_character() ) )
print(head(my_df))

```

```

R> # A tibble: 6 x 2
R>       y z
R>   <dbl> <chr>
R> 1 0.770  a
R> 2 0.185  a
R> 3 0.504  a
R> 4 0.843  a
R> 5 0.976  a
R> 6 0.00403 a

```

O resultado está conforme o esperado, um `dataframe` com duas colunas, a primeira com números e a segunda com texto.

Note que toda exportação com função `write_csv` irá ser formatada, por padrão, com a notação internacional. Caso quiser algo diferentes, verifique as opções disponíveis na função `write_delim`, a qual é muito mais flexível.

### 4.3 Arquivos *Excel* (*xls* e *xlsx*)

Em Finanças e Economia, é bastante comum encontrarmos dados salvos em arquivos do tipo Microsoft Excel, com extensão `.xls` ou `.xlsx`. Apesar de não ser um formato

de armazenamento de dados eficiente, esse é um programa de planilhas bastante popular devido às suas funcionalidades. É muito comum que informações sejam armazenadas e distribuídas dessa forma. Por exemplo, dados históricos do Tesouro Direto são disponibilizados como arquivos *.xls* no site do tesouro nacional. A CVM (Comissão de Valores Mobiliários) e ANBIMA (Associação Brasileira das Entidades dos Mercados Financeiro e de Capitais) também tem preferência por esse tipo de formato em alguns dados publicados em seu *site*.

A desvantagem de usar arquivos do Excel para armazenar dados é sua baixa portabilidade e o maior tempo necessário para leitura e gravação. Isso pode não ser um problema para tabelas pequenas, mas ao lidar com um grande volume de dados, o uso de arquivos Excel é frustrante e não aconselhável. Se possível, evite o uso de arquivos do Excel em seu ciclo de trabalho.

### 4.3.1 Importação de Dados

O R não possui uma função nativa para importar dados do Excel e, portanto, deve-se instalar e utilizar certos pacotes para realizar essa operação. Existem diversas opções, porém, os principais pacotes são `XLConnect` (Mirai Solutions GmbH, 2022), `xlsx` (Dragulescu and Arendt, 2020), `readxl` (Wickham and Bryan, 2022) e `tidyxl` (Garmonsway, 2022).

Apesar de os pacotes anteriores terem objetivos semelhantes, cada um tem suas peculiaridades. Caso a leitura de arquivos do Excel seja algo importante no seu trabalho, aconselho-o fortemente a estudar as diferenças entre esses pacotes. Por exemplo, pacote `tidyxl` permite a leitura de dados não-estruturados de um arquivo Excel, enquanto `XLConnect` possibilita a abertura de uma conexão ativa entre o R e o Excel, onde o usuário pode transmitir dados entre um e o outro, formatar células, criar gráficos no Excel e muito mais.

Nesta seção, daremos prioridade para funções do pacote `readxl`, que é um dos mais fáceis e diretos de se utilizar, além de não necessitar de outros softwares instalados (tal como o *Java*). Para instalar o referido pacote, basta utilizar a função `install.packages`:

```
install.packages('readxl')
```

Imagine agora a existência de um arquivo chamado `Ibov_xls.xlsx` que contenha os mesmos dados do Ibovespa que importamos na seção anterior. A importação das informações contidas nesse arquivo para o R será realizada através da função `read_excel`:

```
library(readxl)
library(dplyr)
```

```

# set file
my_f <- '00-text-resources/data/Ibov_xlsx.xlsx'

# read xlsx into dataframe
my_df <- read_excel(my_f, sheet = 'Sheet1')

# glimpse contents
glimpse(my_df)

```

```

R> Rows: 2,721
R> Columns: 2
R> $ ref.date      <dtm> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price.close <dbl> 70045, 70240, 70729, 70451, 70263, 704~

```

Observe que, nesse caso, as datas já foram importadas com a formatação correta na classe `dtm` (*datetime*). Essa é uma vantagem ao utilizar arquivos do Excel: a classe dos dados do arquivo original é levada em conta no momento da importação. O lado negativo desse formato é a baixa portabilidade dos dados e o maior tempo necessário para a execução da importação. Como regra geral, dados importados do Excel apresentarão um tempo de carregamento mais alto do que dados importados de arquivos `.csv`.

### 4.3.2 Exportação de Dados

A exportação para arquivo Excel também é fácil. Assim como para a importação, não existe uma função nativa do R que execute esse procedimento. Para tal tarefa, temos pacotes `xlsx` e `writexl` (Ooms, 2022). Uma diferença aqui é que o pacote `xlsx` oferece mais funcionalidade mas exige a instalação do Java JDK no sistema operacional. No caso do Windows, basta visitar o site do Java e instalar o software na versão 64 bits (opção *Windows Off-line (64 bits)*). Logo após, instale o pacote `xlsx` normalmente no R com o comando `install.packages('xlsx')`.

Vamos começar com um exemplo para `xlsx`

```

library(xlsx)

# set number of rows
N <- 50

# create random dataframe
my_df <- data.frame(y = seq(1,N),
                   z = rep('a',N))

# write to xlsx

```

```
f_out <- tempfile(fileext = '.xlsx')
write.xlsx(x = my_df,
           file = f_out,
           sheetName = "my df")
```

Note que uma diferença nos argumentos da função `write.xlsx` é que é necessário incluir o nome da aba do arquivo Excel onde os dados da tabela serão exportados. Para exportar várias informações para um mesmo arquivo, é necessário utilizar o argumento `append` da função `write.xlsx`. Caso contrário, a função irá criar um novo arquivo em cada chamada da mesma. Veja o exemplo a seguir, onde exportamos dois dataframes para duas abas diferentes do mesmo arquivo Excel:

```
# set number of rows
N <- 25

# create random dfs
my_df_A <- data.frame(y = seq(1,N),
                     z = rep('a',N))

my_df_B <- data.frame(z = rep('b',N))

# write both df to single file
f_out <- tempfile(fileext = '.xlsx')
write.xlsx(x = my_df_A,
           file = f_out,
           sheetName = "Tabela A")

write.xlsx(x = my_df_B,
           file = f_out,
           sheetName = "Tabela B",
           append = TRUE )
```

Após a exportação, podes verificar as abas disponíveis no arquivo exportado com função `xlsx::getSheets`:

```
readxl::excel_sheets(f_out)
```

```
R> [1] "Tabela A" "Tabela B"
```

O diferencial do pacote `writexl` em relação a `xlsx` é a não necessidade do Java, e a rapidez de execução. O lado negativo é que, na versão atual (1.4.1 – 2022-11-23), não permite a escrita em arquivos já existentes. Veja a seguir:

```

library(writexl)
# set number of rows
N <- 25

# create random dfs
my_df_A <- data.frame(y = seq(1,N),
                     z = rep('a',N))

write_xlsx(x = my_df_A,
           file = f_out)

```

Para comparar o desempenho, vamos verificar a diferença de tempo de execução entre um e outro:

```

library(writexl)
library(readxl)
library(xlsx)

# set number of rows
N <- 2500

# create random dfs
my_df_A <- data.frame(y = seq(1,N),
                     z = rep('a',N))

# set files
my_file_1 <- '00-text-resources/data/temp_writexl.xlsx'
my_file_2 <- '00-text-resources/data/temp_xlsx.xlsx'

# test export
time_write_writexl <- system.time(write_xlsx(x = my_df_A,
                                             path = my_file_1))

time_write_xlsx <- system.time(write.xlsx(x = my_df_A,
                                         file = my_file_2))

# test read
time_read_readxl <- system.time(read_xlsx(path = my_file_1 ))
time_read_xlsx <- system.time(read.xlsx(file = my_file_2,
                                       sheetIndex = 1 ))

```

Após a execução, vamos verificar a diferença de tempo:

```
# results
my_formats <- c('xlsx', 'readxl')
results_read <- c(time_read_xlsx[3], time_read_readxl[3])
results_write<- c(time_write_xlsx[3], time_write_writexl[3])

# print text
my_text <- paste0('\nTime to WRITE dataframe with ',
                  my_formats, ': ',
                  format(results_write, digits = 4),
                  ' seconds', collapse = '')
message(my_text)
```

```
R>
R> Time to WRITE dataframe with xlsx: 1.598 seconds
R> Time to WRITE dataframe with readxl: 0.012 seconds
```

```
my_text <- paste0('\nTime to READ dataframe with ',
                  my_formats, ': ',
                  format(results_read, digits = 4),
                  ' seconds', collapse = '')
message(my_text)
```

```
R>
R> Time to READ dataframe with xlsx: 2.577 seconds
R> Time to READ dataframe with readxl: 0.007 seconds
```

Como podemos ver, mesmo para dados de pouco volume, um dataframe com 2500 linhas e 2 colunas, a diferença de tempo de execução é significativa. Caso estiveres trabalhando com grandes planilhas, o uso de pacotes `readxl` e `writexl` é fortemente recomendado. Porém, como já mostrado anteriormente, as funções de `xlsx` oferecem algumas funcionalidades extras.

## 4.4 Formato *.RData* e *.rds*

O R possui dois formatos nativos para salvar objetos de sua área de trabalho para um arquivo local com extensão *RData* ou *rds*. O grande benefício, em ambos os casos, é que o arquivo resultante é compacto e o seu acesso é muito rápido. A desvantagem é que os dados perdem portabilidade para outros programas. A diferença entre um formato e outro é que arquivos *RData* podem salvar mais de um objeto, enquanto o formato *.rds* salva apenas um. Na prática, porém, essa não é uma restrição forte. No R existe um objeto do tipo *lista* que incorpora outros. Portanto, caso salvarmos uma *lista* em um arquivo *.rds*, podemos gravar no disco quantos objetos forem necessário.

#### 4.4.1 Importação de Dados

Para carregar os dados de um arquivo *RData*, utilizamos a função `load`:

```
# set a object
my_x <- 1:100

# set temp name of RData file
my_file <- adfeR::get_data_file('temp.RData')

# load it
load(file = my_file)
```

O arquivo `temp.RData` possui dois objetos, `my_x` e `my_y`, os quais se tornam disponíveis na área de trabalho depois da chamada de `load`.

O processo de importação para arquivos `.rds` é muito semelhante. A diferença é no uso da função `readr::read_rds`:

```
# set file path
my_file <- adfeR::get_data_file('temp.rds')

# load content into workspace
my_x <- readr::read_rds(file = my_file)
```

Comparando o código entre o uso de arquivos `.RData` e `.rds`, note que um benefício no uso de `.rds` é a explícita definição do objeto na área de trabalho. Isto é, o conteúdo de `my_file` em `readr::read_rds` é explicitamente salvo em `my_x`. Quando usamos a função `load`, no código não fica claro qual o nome do objeto que foi importado. Isso é particularmente inconveniente quando é necessário modificar o nome do objeto importado.



Como sugestão, dê preferência ao uso do formato `.rds`, o qual resulta em códigos mais transparentes. A diferença de velocidade de acesso e gravação entre um e outro é mínima. O benefício de importar vários objetos em um mesmo arquivo com o formato `RData` torna-se irrelevante quando no uso de objetos do tipo lista, os quais podem incorporar outros objetos no seu conteúdo.

#### 4.4.2 Exportação de Dados

Para criar um novo arquivo *RData*, utilizamos a função `save`. Veja o exemplo a seguir, onde criamos um arquivo *RData* com dois objetos:

```
# set vars
my_x <- 1:100
my_y <- 1:100

# write to RData
my_file <- tempfile(fileext = '.RData')
save(list = c('my_x', 'my_y'),
      file = my_file)
```

Podemos verificar a existência do arquivo com a função `file.exists`:

```
file.exists(my_file)
```

```
R> [1] TRUE
```

Observe que o arquivo `file11cf1c670b24.RData` está disponível na pasta temporária.

Já para arquivos `.rds`, salvamos o objeto com função `saveRDS`:

```
# set data and file
my_x <- 1:100
my_file <- '00-text-resources/data/temp.rds'

# save as .rds
saveRDS(object = my_x,
        file = my_file)

# read it
my_x2 <- readRDS(file = my_file)

# test equality
print(identical(my_x, my_x2))
```

```
R> [1] TRUE
```

O comando `identical` testa a igualdade entre os objetos e, como esperado, verificamos que `my_x` e `my_x2` são exatamente iguais.

## 4.5 Arquivos *fst* (pacote *fst*)

O formato *fst* foi especialmente desenhado para possibilitar a gravação e leitura de dados tabulares de forma rápida e com mínimo uso do espaço no disco. O uso deste formato é particularmente benéfico quando se está trabalhando com volumosas bases de dados em computadores potentes. **O grande truque do formato *fst* é usar todos núcleos do computador para importar e exportar dados**, enquanto

todos os demais formatos se utilizam de apenas um. Como logo veremos, o ganho em velocidade é bastante significativo.

### 4.5.1 Importação de Dados

O uso do formato *fst* é bastante simples. Utilizamos a função `read_fst` para ler arquivos:

```
library(fst)

my_file <- adfeR::get_data_file('temp.fst')
my_df <- read_fst(my_file)

glimpse(my_df)
```

```
R> Rows: 1,000
R> Columns: 1
R> $ x <dbl> 0.70968891, 0.83903044, 0.70026554, 0.78120026, ~
```

Assim como para os demais casos, os dados estão disponíveis na área de trabalho após a importação.

### 4.5.2 Exportação de Dados

Utilizamos a função `write_fst` para gravar arquivos no formato *fst*, :

```
library(fst)

# create dataframe
N <- 1000
my_file <- tempfile(fileext = '.fst')
my_df <- data.frame(x = runif(N))

# write to fst
write_fst(x = my_df, path = my_file)
```

### 4.5.3 Testando o Tempo de Execução do Formato *fst*

Como um teste do potencial do pacote `fst`, a seguir vamos cronometrar o tempo de leitura e gravação entre *fst* e *rds* para um dataframe com grande quantidade de dados: 5,000,000 linhas e 2 colunas. Iremos reportar também o tamanho do arquivo resultante.

```
library(fst)
```

```

# set number of rows
N <- 5000000

# create random dfs
my_df <- data.frame(y = seq(1,N),
                   z = rep('a',N))

# set files
my_file_1 <- '00-text-resources/data/temp_rds.rds'
my_file_2 <- '00-text-resources/data/temp_fst.fst'

# test write
time_write_rds <- system.time(write_rds(my_df, my_file_1 ))
time_write_fst <- system.time(write_fst(my_df, my_file_2 ))

# test read
time_read_rds <- system.time(readRDS(my_file_1))
time_read_fst <- system.time(read_fst(my_file_2))

# test file size (MB)
file_size_rds <- file.size(my_file_1)/1000000
file_size_fst <- file.size(my_file_2)/1000000

```

Após a execução, vamos verificar o resultado:

```

# results
my_formats <- c('.rds', '.fst')
results_read <- c(time_read_rds[3], time_read_fst[3])
results_write <- c(time_write_rds[3], time_write_fst[3])
results_file_size <- c(file_size_rds , file_size_fst)

# print text
my_text <- paste0('\nTime to WRITE dataframe with ',
                 my_formats, ': ',
                 results_write, ' seconds', collapse = '')
message(my_text)

```

R>

R> Time to WRITE dataframe with .rds: 1.1579999999999 seconds

R> Time to WRITE dataframe with .fst: 0.09299999999998472 seconds

```

my_text <- paste0('\nTime to READ dataframe with ',
                 my_formats, ': ',

```

```

      results_read, ' seconds', collapse = '')
message(my_text)

R>
R> Time to READ dataframe with .rds: 1.01900000000001 seconds
R> Time to READ dataframe with .fst: 0.096999999999998 seconds

my_text <- paste0('\nResulting FILE SIZE for ',
                  my_formats, ': ',
                  results_file_size, ' MBs', collapse = '')
message(my_text)

R>
R> Resulting FILE SIZE for .rds: 65.01011 MBs
R> Resulting FILE SIZE for .fst: 14.791938 MBs

```

A diferença é gritante! O formato *fst* não somente lê e grava com mais rapidez mas o arquivo resultante também é menor. Porém, saiba que os resultados anteriores foram compilados em um computador com 16 núcleos. É possível que a diferença de tempo para um computador mais modesto não seja tão significativa.



Devido ao uso de todos os núcleos do computador, o formato *fst* é altamente recomendado quando estiver trabalhando com dados volumosos em um computador potente. Não somente os arquivos resultantes serão menores, mas o processo de gravação e leitura será consideravelmente mais rápido.

## 4.6 Arquivos *SQLite*

O uso de arquivos *csv*, *rds* e *fst* para armazenar conjuntos de dados tem seus limites a medida que o tamanho dos arquivos aumenta e os dados fragmentam-se em várias tabelas. Se você está esperando muito tempo para ler apenas uma tabela de um arquivo com várias tabelas, deve procurar alternativas mais eficientes. Da mesma forma, se você estiver trabalhando em uma rede de computadores e muitas pessoas estão usando os mesmos dados, faz sentido manter e distribuir as informações de um servidor central. Dessa forma, cada usuário pode ter acesso à mesma informação, simultaneamente.

Isso nos leva ao tópico de programas de armazenamento e distribuição de banco de dados. Esses programas específicos geralmente funcionam com uma linguagem de consulta chamada *SQL* (*Structured Query Language*), e permitem ao usuário ler partes dos dados e mesmo manipulá-lo de forma eficiente. Existem muitas opções de software de banco de dados que se integra muito bem com R. A lista inclui

**mySQL, SQLite e MariaDB.** Aqui, forneceremos um tutorial rápido sobre esse tópico usando o SQLite, que é o mais fácil de usar, uma vez que não precisa de nenhuma configuração do servidor e todos dados estão contidos em um único arquivo.

Antes de irmos para os exemplos, precisamos entender como se usa o *software* de banco de dados. Primeiro, um banco de dados deve existir em seu computador ou rede. Segundo, o R se conectará ao banco de dados e retornará um objeto de conexão. Com base nessa conexão, enviaremos consultas para importar dados desse banco de dados usando a linguagem *SQL*. A principal vantagem é que podemos ter um grande banco de dados de, digamos, 10 GB e carregar apenas uma pequena porção dele na área de trabalho do R. Essa operação também é muito rápida, permitindo um acesso eficiente às tabelas disponíveis.

#### 4.6.1 Importação de Dados

Assumindo a existência de um arquivo com formato SQLite, podemos importar suas tabelas com o pacote `RSQLite`:

```
library(RSQLite)

# set name of SQLITE file
f_sqlite <- adfer::get_data_file('SQLite_db.SQLITE')

# open connection
my_con <- dbConnect(drv = SQLite(), f_sqlite)

# read table
my_df <- dbReadTable(conn = my_con,
                     name = 'MyTable1') # name of table in sqlite

# print with str
glimpse(my_df)
```

```
R> Rows: 1,000,000
R> Columns: 2
R> $ x <dbl> 0.007504194, 0.439465174, 0.178387480, 0.9857759~
R> $ G <chr> "B", "B", "B", "B", "A", "B", "A", "B", "B", "B"~
```

Outro exemplo do uso do `SQLITE` é com instruções de um comando *SQL*. Observe que, no código anterior, usamos função `dbReadTable` para obter o conteúdo de todas as linhas da tabela `MyTable1`. Agora, vamos usar o comando `dbGetQuery` para obter dados da tabela `myTable2` apenas quando a coluna `G` é igual a `A`:

```

# set sql statement
my_SQL_statement <- "select * from myTable2 where G='A'"

# get query
my_df_A <- dbGetQuery(conn = my_con,
                      statement = my_SQL_statement)

# disconnect from db
dbDisconnect(my_con)

# print with str
print(str(my_df_A))

```

```

R> 'data.frame':   499522 obs. of  2 variables:
R> $ x: num  0.0637 0.1982 0.2894 0.7389 0.0669 ...
R> $ G: chr  "A" "A" "A" "A" ...
R> NULL

```

Nesse exemplo simples podemos ver como é fácil criar uma conexão com um banco de dados, recuperar tabelas e desconectar. Se você estiver trabalhando com dados volumosos e diversas tabelas, vale a pena utilizar um software de banco de dados apropriado. Caso existir um servidor de banco de dados disponível em seu local de trabalho, eu recomendo fortemente aprender a conectar-se a ele e carregar dados diretamente de uma sessão do R.

#### 4.6.2 Exportação de Dados

Como exemplo, vamos criar dois dataframes com dados aleatórios e salvar ambos em um arquivo SQLite usando pacote RSQLite.

```

library(RSQLite)

# set number of rows in df
N = 10^6

# create simulated dataframe
my_large_df_1 <- data.frame(x=runif(N),
                           G= sample(c('A','B'),
                                     size = N,
                                     replace = TRUE))

my_large_df_2 <- data.frame(x=runif(N),
                           G = sample(c('A','B'),

```

```

                                size = N,
                                replace = TRUE))

# set name of SQLITE file
f_sqlite <- tempfile(fileext = '.SQLITE')

# open connection
my_con <- dbConnect(drv = SQLite(), f_sqlite)

# write df to sqlite
dbWriteTable(conn = my_con, name = 'MyTable1',
              value = my_large_df_1)
dbWriteTable(conn = my_con, name = 'MyTable2',
              value = my_large_df_2)

# disconnect
dbDisconnect(my_con)

```

A saída TRUE de `dbWriteTable` indica que tudo ocorreu bem. Uma conexão foi aberta usando a função `dbConnect` e os dataframes foram escritos em um arquivo SQLITE temporário chamado `file11cf7e9dbfad.SQLITE`. É boa política de programação sempre se desconectar do banco de dados após a utilização. Fizemos isso com a função `dbDisconnect`.

## 4.7 Dados Não-Estruturados e Outros Formatos

Os pacotes e formatos anteriores são suficientes para resolver o problema de importação de dados na grande maioria das situações. Apesar disso, vale destacar que o R possui outras funções específicas para diferentes formatos. Isso inclui arquivos exportados de outros softwares, tal como SPSS, Matlab, entre vários outros. Se esse for o seu caso, sugiro um estudo aprofundado do pacote `foreign` (R Core Team, 2022).

Em alguns casos nos deparamos com dados armazenados de uma forma não estruturada, tal como um texto qualquer. Pode-se importar o conteúdo de um arquivo de texto linha por linha através da função `readr::read_lines`. Veja o exemplo a seguir, onde importamos o conteúdo inteiro do livro *Pride and Prejudice*:

```

# set file to read
my_f <- adfeR::get_data_file('pride_and_prejudice.txt')

# read file line by line
my_txt <- read_lines(my_f)

```

```
# print 50 characters of first fifteen lines
print(str_sub(string = my_txt[1:15],
             start = 1,
             end = 50))
```

```
R> [1] "The Project Gutenberg EBook of Pride and Prejudice"
R> [2] ""
R> [3] "This eBook is for the use of anyone anywhere at no"
R> [4] "almost no restrictions whatsoever. You may copy i"
R> [5] "re-use it under the terms of the Project Gutenberg"
R> [6] "with this eBook or online at www.gutenberg.org"
R> [7] ""
R> [8] ""
R> [9] "Title: Pride and Prejudice"
R> [10] ""
R> [11] "Author: Jane Austen"
R> [12] ""
R> [13] "Posting Date: August 26, 2008 [EBook #1342]"
R> [14] "Release Date: June, 1998"
R> [15] "Last Updated: March 10, 2018"
```

Neste exemplo, arquivo `pride_and_prejudice.txt` contém todo o conteúdo do livro *Pride and Prejudice* de Jane Austen, disponível gratuitamente pelo projeto Gutenberg<sup>3</sup>. Importamos todo o conteúdo do arquivo como um vetor de texto denominado `my_txt`. Cada elemento de `my_txt` é uma linha do arquivo do texto original. Com base nisso, podemos calcular o número de linhas do livro e o número de vezes que o nome 'Bennet', um dos protagonistas, aparece no texto:

```
# count number of lines
n_lines <- length(my_txt)

# set target text
name_to_search <- 'Bennet'

# set function for counting words
fct_count_bennet <- function(str_in, target_text) {

  require(stringr)

  n_words <- length(str_locate_all(string = str_in,
```

---

<sup>3</sup><http://www.gutenberg.org/>

```

                                pattern = target_text)[[1]])

    return(n_words)
}

# use fct for all lines of Pride and Prejudice
n_times <- sum(sapply(X = my_txt,
                     FUN = fct_count_bennet,
                     target_text = name_to_search))

# print results
my_msg <- paste0('The number of lines found in the file is ',
                 n_lines, '\n',
                 'The word "', name_to_search, '" appears ',
                 n_times, ' times in the book.')
message(my_msg)

```

R> The number of lines found in the file is 13427.

R> The word "Bennet" appears 664 times in the book.

No exemplo, mais uma vez usamos `sapply`. Neste caso, a função nos permitiu usar outra função para cada elemento de `my_txt`. Neste caso, procuramos e contamos o número de vezes que a palavra “Bennet” foi encontrada no texto. Observe que poderíamos simplesmente mudar `name_to_search` por qualquer outro nome, caso quiséssemos.

#### 4.7.1 Exportando de Dados Não-Estruturados

Em algumas situações, é necessário exportar algum tipo de texto para um arquivo. Por exemplo: quando se precisa salvar o registro de um procedimento em um arquivo de texto; ou quando se precisa gravar informações em um formato específico não suportado pelo R. Esse procedimento é bastante simples. Junto à função `readr::write_lines`, basta indicar um arquivo de texto para a saída com o argumento `file`. Veja a seguir:

```

# set file
my_f <- tempfile(fileext = '.txt')

# set some string
my_text <- paste0('Today is ', Sys.Date(), '\n',
                  'Tomorrow is ', Sys.Date()+1)

# save string to file

```

```
write_lines(x = my_text, file = my_f, append = FALSE)
```

No exemplo, criamos um objeto de texto com uma mensagem sobre a data atual e gravamos a saída no arquivo temporário `file11cf3fe46972.txt`. Podemos checar o resultado com a função `readr::read_lines`:

```
print(read_lines(my_f))
```

```
R> [1] "Today is 2022-11-23"      "Tomorrow is 2022-11-24"
```

## 4.8 Selecionando o Formato

Após entendermos a forma de salvar e carregar dados de arquivos locais em diferentes formatos, é importante discutirmos sobre a escolha do formato. O usuário deve levar em conta três pontos nessa decisão:

- velocidade de importação e exportação;
- tamanho do arquivo resultante;
- compatibilidade com outros programas e sistemas.

Na grande maioria das situações, o uso de arquivos *csv* satisfaz esses quesitos. Ele nada mais é do que um arquivo de texto que pode ser aberto, visualizado e importado em qualquer programa. Desse modo, fica muito fácil compartilhar dados compatíveis com outros usuários. Além disso, o tamanho de arquivos *csv* geralmente não é exagerado em computadores modernos. Caso necessário, podemos compactar o arquivo *.csv* usando o programa *7zip*, por exemplo, o qual irá diminuir consideravelmente o tamanho do arquivo. Por esses motivos, **o uso de arquivos *csv* para importações e exportações é preferível na grande maioria das situações.**

Existem casos, porém, onde a velocidade de importação e exportação pode fazer diferença. Caso abrir mão de portabilidade não faça diferença ao projeto, o formato *rds* é ótimo e prático. Se este não foi suficiente, então a melhor alternativa é partir para o *fst*, o qual usa maior parte do *hardware* do computador para importar os dados. Como sugestão, caso puder, apenas **evite o formato do Excel**, o qual é o menos eficiente de todos.

## 4.9 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Crie um dataframe com o código a seguir:

```
library(dplyr)
```

```
my_N <- 10000
```

```
my_df <- tibble(x = 1:my_N,  
               y = runif(my_N))
```

Exporte o dataframe resultante para cada um dos cinco formatos: csv, rds, xlsx, fst. Qual dos formatos ocupou maior espaço na memória do computador? Dica: `file.size` calcula o tamanho de arquivos dentro do próprio R.

02 - Melhore o código anterior com a mensuração do tempo de execução necessário para gravar os dados nos diferentes formatos. Qual formato teve a gravação mais rápida? Dica: use função `system.time` ou pacote `tictoc` para calcular os tempos de execução.

03 - Para o código anterior, redefina o valor de `my_N` para 1000000. Esta mudança modifica as respostas das duas últimas perguntas?

04 - Use função `adfeR::get_data_file` para acessar o arquivo `SP500.csv` no repositório de dados do livro. Importe o conteúdo do arquivo no R com função `readr::read_csv`. Quantas linhas existem no dataframe resultante?

05 - No link <https://eeecon.uibk.ac.at/~zeileis/grunfeld/Grunfeld.csv> você encontrará um arquivo `.csv` para os dados *Grunfeld*. Esta é uma tabela particularmente famosa devido ao seu uso como dados de referência em modelos econométricos. Usando função `readr::read_csv`, leia este arquivo usando o link direto como entrada em `read_csv`. Quantas colunas você encontra no dataframe resultante?

06 - Use função `adfeR::get_data_file` para acessar o arquivo `example_tsv.csv` no repositório de dados do livro. Note que as colunas dos dados estão separadas pelo símbolo de tabulação (`'\t'`). Após ler o manual do `readr::read_delim`, importe as informações deste arquivo para o seu computador. Quantas linhas o arquivo contém?

07 - No pacote do livro existe um arquivo de dados chamado `'funky_csv2.csv'`. Este possui um formato particularmente bizarro para os dados. Abra o mesmo em um editor de texto e procure entender como as colunas são separadas e qual o símbolo para o decimal. Após isso, veja as entradas da função `read.table` e importe a tabela na sessão do R. Caso somarmos o número de linhas com o número de colunas da tabela importada, qual o resultado?



# Capítulo 5

## Importação de Dados via Pacotes

Uma das grandes vantagens de se utilizar o R é a quantidade de dados que podem ser importados através da internet. Isso é especialmente prático pois uma base de dados pode ser atualizada através de um simples comando, evitando o tedioso trabalho de coleta manual. Ao usarmos pacotes para importar dados, esta etapa da pesquisa se torna reproduzível e mais rápida, facilitando o compartilhamento e futura execução do nosso código.

Neste capítulo vou descrever e dar exemplos de importação de dados para os mais importantes e estáveis pacotes especializados na importação de para dados financeiros e econômicos no Brasil e exterior. A lista inclui:

**GetQuandlData (Perlin, 2019)** Importa dados econômicos e financeiros do vasto repositório de dados da plataforma *Quandl*.

**BatchGetSymbols (Perlin, 2022a)** Importa dados de preços diários de ações e índices do *Yahoo Finance*.

**GetTDDData (Perlin, 2022c)** Importa dados de títulos de dívida pública do Brasil diretamente do site do Tesouro Direto.

**GetBCBData (Perlin, 2022b)** Importa dados do grande repositório de séries temporais do Banco Central do Brasil, local obrigatório para qualquer economista que trabalha com dados.

**GetDFPData2 (Perlin and Kirch, 2022a)** Importa dados do sistema DFP – Demonstrativos Financeiros Padronizados – de empresas negociadas na B3, a bolsa Brasileira. O repositório inclui documentos financeiros tal como o balanço patrimonial, demonstrativos de resultados, entre vários outros.

**GetFREData (Perlin and Kirch, 2022b)** Importa dados do sistema FRE – Formulário de Referência – da bolsa Brasileira. Esta inclui diversos eventos e informações corporativas tal como composição do conselho e diretoria, remu-

neração dos conselheiros, entre outras.

## 5.1 Pacote `GetQuandlData`

*Quandl* é um repositório de dados abrangente, fornecendo acesso a uma série de tabelas gratuitas e pagas disponibilizadas por diversas instituições de pesquisa. Como ponto inicial, recomendo fortemente que você navegue nas tabelas disponíveis no site da *Quandl*<sup>1</sup>. Verás que uma grande proporção dos repositórios de dados abertos em economia e finanças também estão disponíveis no *Quandl*.

Pacote `Quandl` (Raymond McTaggart et al., 2021) é a extensão oficial oferecida pela empresa e disponível no CRAN. No entanto, o pacote tem alguns problemas quanto a estrutura de dados de saída<sup>2</sup> e, para resolver, escrevi o meu próprio pacote `GetQuandlData` (Perlin, 2019). O diferencial de `GetQuandlData` é a saída de dados já estruturados, prontos para uma posterior análise.

A **primeira e obrigatória** etapa no uso de `GetQuandlData` é registrar o usuário no site. Logo em seguida, vá para *Account Settings* e procure pela seção *You API Key*. Este local deve mostrar uma senha, tal como `Asv8Ac7zuZzJSCGxynfG`. Copie o texto para a área de transferência (*control + c*) e, no R, defina um objeto de contendo o conteúdo copiado da seguinte maneira:

```
# set FAKE api key to quandl
my_api_key <- 'Asv8Ac7zuZzJSCGxynfG'
```

Essa chave API é exclusiva para cada usuário e a apresentada aqui **não funcionará no seu computador**. Você precisará obter sua própria chave de API para executar os exemplos do livro. Depois de encontrar e definir sua chave, vá para o site do *Quandl* e use a caixa de pesquisa para procurar o símbolo da série temporal de interesse.

Como exemplo, usaremos dados do preço do ouro no mercado Londrino. O código para esta série no *Quandl* é `'LBMA/GOLD'`. Observe que a estrutura de um identificador de tabelas no *Quandl* é sempre a mesma, com o nome do banco de dados primeiro e o nome da tabela depois, separados por uma barra (/).

Agora, com a chave API e o identificador da tabela, usamos a função `get_Quandl_series` para baixar os dados de 1980-01-01 a 2021-01-01:

```
library(GetQuandlData)
library(tidyverse)

# set symbol and dates
```

<sup>1</sup><https://www.quandl.com/>

<sup>2</sup>Veja postagem no blog

```

my_symbol <- c('GOLD' = 'LBMA/GOLD')
first_date <- '1980-01-01'
last_date <- '2021-01-01'

# get data!
df_quandl <- get_Quandl_series(id_in = my_symbol,
                              api_key = my_api_key,
                              first_date = first_date,
                              last_date = last_date)

# check it
glimpse(df_quandl)

```

```

R> Rows: 10,763
R> Columns: 9
R> $ `USD (AM)` <chr> "1766.6", "1772.9", "1766.75", "1758.9~
R> $ `USD (PM)` <chr> "1761.25", "1779.75", "1772.4", "1753.~
R> $ `GBP (AM)` <chr> "1450.03", "1452.36", "1443.26", "1447~
R> $ `GBP (PM)` <chr> "1451.62", "1457.26", "1444.86", "1451~
R> $ `EURO (AM)` <chr> "1734.09", "1732.3", "1722.23", "1724.~
R> $ `EURO (PM)` <chr> "1735.54", "1743.62", "1727.91", "1725~
R> $ series_name <chr> "GOLD", "GOLD", "GOLD", "GOLD", "GOLD"~
R> $ ref_date <date> 2022-08-03, 2022-08-02, 2022-08-01, 2~
R> $ id_quandl <chr> "LBMA/GOLD", "LBMA/GOLD", "LBMA/GOLD",~

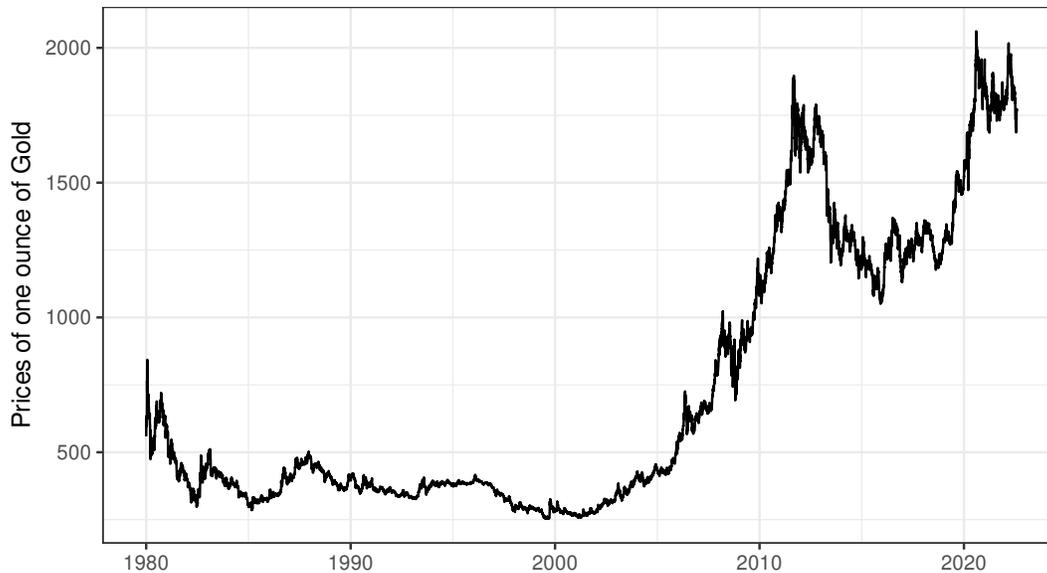
```

Observe como definimos o nome da série temporal em linha `id_in = c('GOLD' = 'LBMA/GOLD')`. O nome do elemento – GOLD – se torna uma coluna chamada `series_name` no dataframe de saída. Se tivéssemos mais séries temporais, elas seriam empilhadas na mesma tabela.

Para verificar os dados, vamos criar um gráfico com os preços do ouro ao longo do tempo. Aqui, usaremos o pacote `ggplot2` para criar a figura. Por enquanto você não precisa se preocupar com o código de criação de gráficos. Teremos o capítulo 10 inteiro dedicado ao tópico.

## Gold Price: London Fixing - USD

1980-01-01 to 2021-01-01



Data from Quandl -- London Bullion Market Association

De modo geral, os preços do ouro permaneceram relativamente estáveis entre 1980 e 2000, atingindo um pico após 2010. Uma possível explicação é a maior demanda por ativos mais seguros, como o ouro, após a crise financeira de 2009. No entanto, o ouro nunca foi um investimento eficiente a longo prazo. Para mostrar isso, vamos calcular seu retorno anual composto de 1980-01-02 a 2022-08-03:

```
# sort the rows
df_quandl <- df_quandl %>%
  mutate(USD = as.numeric(`USD (AM)`) ) %>%
  arrange(ref_date)

total_ret <- last(df_quandl$USD)/first(df_quandl$USD) - 1
total_years <- as.numeric(max(df_quandl$ref_date) -
  min(df_quandl$ref_date) )/365

comp_ret_per_year <- (1 + total_ret)^(1/total_years) - 1

print(comp_ret_per_year)
```

```
R> [1] 0.02737004
```

Encontramos o resultado de que os preços do ouro em USD apresentam um retorno composto equivalente a 2,74% ao ano. Este não é um resultado de investimento

impressionante de forma alguma. Como comparação, a inflação anual para os EUA no mesmo período é de 3,22% ao ano. Isso significa que, ao comprar ouro em 1980, o investidor recebeu menos que a inflação como retorno nominal, resultando em perda de poder de compra.

### 5.1.1 Importando Múltiplas Séries

Ao solicitar várias séries temporais do Quandl, pacote `GetQuandlData` empilha todos os dados em um único `dataframe`, tornando mais fácil trabalhar com as ferramentas do `tidyverse`. Como exemplo, vamos olhar para o banco de dados `RATEINF`, o qual contém séries temporais das taxas de inflação ao redor do mundo. Primeiro, precisamos ver quais são os conjuntos de dados disponíveis:

```
library(GetQuandlData)
library(tidyverse)

# databse to get info
db_id <- 'RATEINF'

# get info
df_db <- get_database_info(db_id, my_api_key)

glimpse(df_db)
```

```
R> Rows: 26
R> Columns: 8
R> $ code      <chr> "CPI_ARG", "CPI_AUS", "CPI_CAN", "CPI~
R> $ name      <chr> "Consumer Price Index - Argentina", "~
R> $ description <chr> "Please visit <a href=http://www.rate~
R> $ refreshed_at <dtm> 2020-10-10 02:03:32, 2022-11-19 02:0~
R> $ from_date  <date> 1988-01-31, 1948-09-30, 1989-01-31, ~
R> $ to_date    <date> 2013-12-31, 2022-09-30, 2022-10-31, ~
R> $ quandl_code <chr> "RATEINF/CPI_ARG", "RATEINF/CPI_AUS",~
R> $ quandl_db  <chr> "RATEINF", "RATEINF", "RATEINF", "RAT~
```

Coluna `name` contém a descrição das tabelas com os seguintes nomes:

```
print(unique(df_db$name))

R> [1] "Consumer Price Index - Argentina"
R> [2] "Consumer Price Index - Australia"
R> [3] "Consumer Price Index - Canada"
R> [4] "Consumer Price Index - Switzerland"
R> [5] "Consumer Price Index - Germany"
R> [6] "Consumer Price Index - Euro Area"
```



```

first_date = first_date,
last_date = last_date)

glimpse(df_inflation)

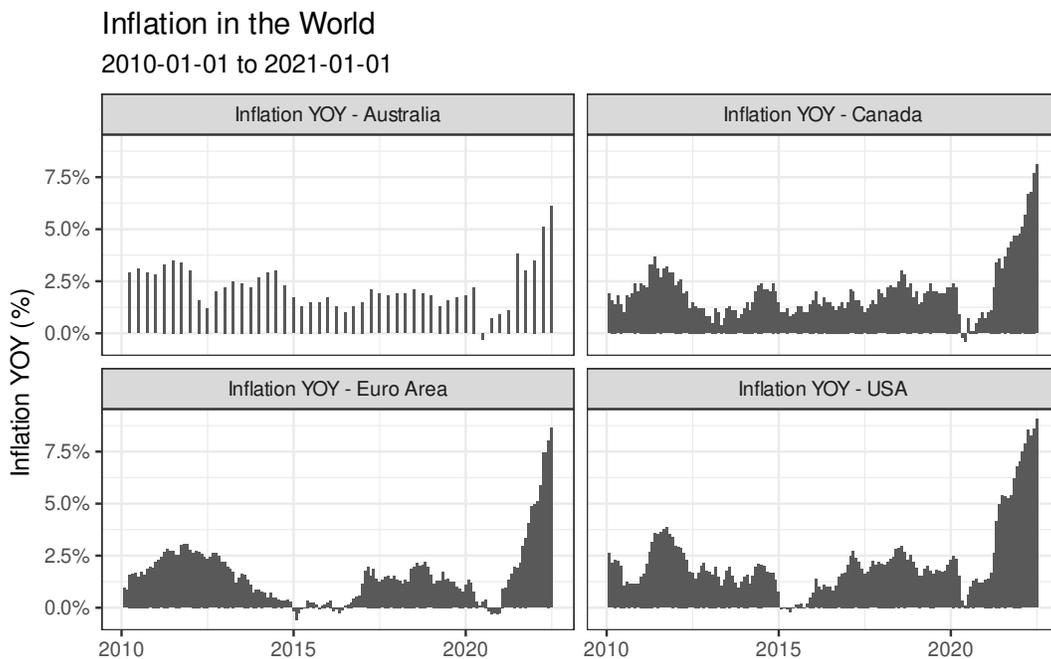
```

```

R> Rows: 500
R> Columns: 4
R> $ series_name <chr> "Inflation YOY - Australia", "Inflatio~
R> $ ref_date <date> 2022-06-30, 2022-03-31, 2021-12-31, 2~
R> $ value <dbl> 6.1, 5.1, 3.5, 3.0, 3.8, 1.1, 0.9, 0.7~
R> $ id_quandl <chr> "RATEINF/INFLATION_AUS", "RATEINF/INFL~

```

Por fim, construímos um gráfico para visualizar as séries de inflação:



Data from Quandl

Como podemos ver, com algumas linhas de código do R conseguimos importar dados atualizados da inflação de diferentes regiões do mundo. O retorno de um `dataframe` no formato empilhado facilitou o processo de análise pois essa é a estrutura de dados que o `ggplot2` espera.

## 5.2 Pacote BatchGetSymbols

Pacote `BatchGetSymbols` faz a comunicação do R com os dados financeiros disponíveis no *Yahoo Finance*. Essa gigantesca base de dados inclui valores agregados de preços e volumes negociados de ações na B3 e outras bolsas internacionais na frequência diária. Tudo que se precisa saber para acessar a base de dados são os identificadores das ações (*tickers*) e um período de tempo.

Os diferenciais do `BatchGetSymbols` são:

**Limpeza e organização:** todos os dados financeiros de diferentes *tickers* são mantidos no mesmo `dataframe`, facilitando a análise futura com as ferramentas do `tidyverse`.

**Controle de erros de importação:** todos erros de *download* são registrados na saída do programa. Caso uma ação em particular não exista no *Yahoo Finance*, esta será ignorada e apenas as demais disponíveis serão retornadas na saída do código;

**Comparação de datas a um *benchmark*:** os dados de ativos individuais são comparados com dados disponíveis para um ativo *benchmark*, geralmente um índice de mercado. Caso o número de datas faltantes seja maior que um determinado limite imposto pelo usuário, a ação é retirada do `dataframe` final.

**Uso de sistema de cache:** no momento de acesso aos dados, os mesmos são salvos localmente no computador do usuário e são persistentes para cada sessão. Caso o usuário requisitar os mesmos dados na mesma sessão do R, o sistema de cache será utilizado. Se os dados desejados não estão disponíveis no cache, a função irá comparar e baixar apenas as informações que faltam. Isso aumenta significativamente a velocidade de acesso aos dados, ao mesmo tempo em que minimiza o uso da conexão a Internet;



Desde versão 2.6 (2020-11-22) de `BatchGetSymbols` a pasta *default* de cache do `BatchGetSymbols` se localiza no diretório temporário da sessão do R. Assim, o cache é persistente apenas para a sessão do usuário. Esta mudança foi motivada por quebras estruturais nos dados do *Yahoo Finance*, onde os dados passados registrados em cache não mais estavam corretos devido a eventos corporativos. O usuário, porém, pode trocar a pasta de cache usando a entrada `cache.folder`.

**Acesso a *tickers* em índices de mercado:** O pacote inclui funções para baixar a composição dos índices Ibovespa, SP500 e FTSE100. Isso facilita a importação de dados para uma grande quantidade de ações. Podes, por exemplo, baixar cotações de todas as ações que fazem parte de certo índice.



A saída de `BatchGetSymbols` é um objeto do tipo lista, ainda não visto no livro. Por enquanto, tudo que precisas saber é que uma lista é um objeto flexível, acomodando outros objetos em sua composição. O acesso a cada elemento de uma lista pode ser feito pelo operador `$`. No capítulo 6 iremos estudar melhor esta classe de objetos.



Note que as entradas da função `BatchGetSymbols::BatchGetSymbols` usam o “.” em seus nomes, tal como `thresh.bad.data`, e `bench.ticker`, enquanto o livro está escrito usando o traço baixo (`_`), tal como `thresh_bad_data`, e `bench_ticker`. Esta diferença pode resultar em problemas se, na falta de atenção, o usuário trocar um pelo outro. Como regra, procure dar prioridade para o uso de traço baixo nos nomes de objetos. Infelizmente algumas funções escritas no passado acabaram ficando com a estrutura antiga e, para não prejudicar os usuários, os nomes das entradas foram mantidos.

Voltando ao nosso exemplo, função `BatchGetSymbols` retorna uma lista com dois elementos: um `dataframe` com o resultado do processo de importação – `df_control` – e outro `dataframe` com os dados das ações – `df_tickers`. Vamos checar o conteúdo do primeiro `dataframe`.

```
# print result of download process
print(l_out$df.control)
```

```
R> # A tibble: 4 x 6
R>   ticker   src  download.status total.obs perc.be~1 thres~2
R>   <chr>   <chr> <chr>                <int>     <dbl> <chr>
R> 1 PETR4.SA yahoo OK                    246         1 KEEP
R> 2 CIEL3.SA yahoo OK                    246         1 KEEP
R> 3 GGBR4.SA yahoo OK                    246         1 KEEP
R> 4 GOAU4.SA yahoo OK                    246         1 KEEP
R> # ... with abbreviated variable names
R> #   1: perc.benchmark.dates, 2: threshold.decision
```

Objeto `df.control` mostra que todos *tickers* foram válidos, com um total de 246 observações para cada ativo. Note que as datas batem 100% com o Ibovespa (coluna `perc.benchmark.dates`).

Quanto aos dados financeiros, esses estão contidos em `l_out$df.tickers`:

```
# print df_tickers
glimpse(l_out$df.tickers)
```

```
R> Rows: 984
R> Columns: 10
```

```
R> $ price.open          <dbl> 29.36, 29.30, 29.84, 26.99, 28~
R> $ price.high         <dbl> 29.82, 29.64, 30.67, 28.49, 28~
R> $ price.low          <dbl> 28.81, 28.86, 29.48, 26.20, 28~
R> $ price.close        <dbl> 29.47, 29.43, 29.60, 28.36, 28~
R> $ volume             <dbl> 84672200, 129384600, 135078600~
R> $ price.adjusted     <dbl> 15.33113, 15.31032, 15.39875, ~
R> $ ref.date           <date> 2021-11-29, 2021-11-30, 2021--
R> $ ticker             <chr> "PETR4.SA", "PETR4.SA", "PETR4~
R> $ ret.adjusted.prices <dbl> NA, -0.0013573042, 0.005776432~
R> $ ret.closing.prices <dbl> NA, -0.0013572786, 0.005776418~
```

Como esperado, a informação sobre preços, retornos e volumes está lá, com as devidas classes de colunas: *dbl* (*double*) para valores numéricos e *date* para as datas. Observe que uma coluna chamada *ticker* também está incluída. Essa indica em que linhas da tabela os dados de uma ação começam e terminam. Mais tarde, no capítulo 9, usaremos essa coluna para fazer diversos cálculos para cada ação.

### 5.2.1 Baixando Dados da Composição do Ibovespa

Outra função útil do pacote é `BatchGetSymbols::GetIbovStocks`, a qual importa a composição atual do índice Ibovespa diretamente do site da B3. Esse índice é um termômetro do mercado local e as ações que o compõem são selecionadas devido sua alta negociabilidade. Portanto, sequenciando o uso de `GetIbovStocks` e `BatchGetSymbols`, podemos facilmente baixar uma volumosa quantidade de dados de ações para o mercado Brasileiro. Considere o seguinte fragmento de código, onde realizamos essa operação:

```
library(BatchGetSymbols)

# set tickers
df_ibov <- GetIbovStocks()
my_tickers <- paste0(df_ibov$tickers, '.SA')

# set dates and other inputs
first_date <- Sys.Date()-30
last_date <- Sys.Date()
thresh_bad_data <- 0.95 # sets percent threshold for bad data
bench_ticker <- '^BVSP' # set benchmark as ibovespa
cache_folder <- 'data/BGS_Cache' # set folder for cache

l_out <- BatchGetSymbols(tickers = my_tickers,
                        first.date = first_date,
```

```
last.date = last_date,
bench.ticker = bench_ticker,
thresh.bad.data = thresh_bad_data,
cache.folder = cache_folder)
```

Note que utilizamos a função `paste0` para adicionar o texto `'.SA'` para cada *ticker* em `df_ibov$tickers`. A saída do código anterior não foi mostrada para não encher páginas e páginas com as mensagens do processamento. Destaco que, caso necessário, poderíamos facilmente exportar os dados em `l_out` para um arquivo `.rds` e futuramente carregá-los localmente para realizar algum tipo de análise.



Saiba que **os preços do Yahoo Finance não são ajustados a dividendos**. O ajuste realizado pelo sistema é apenas para desdobramentos das ações. Isso significa que, ao olhar séries de preços em um longo período, existe um viés de retorno para baixo. Ao comparar com outro software que faça o ajustamento dos preços por dividendos, verá uma grande diferença na rentabilidade total das ações. Como regra, em uma pesquisa formal, **evite usar dados de ações individuais no Yahoo Finance para períodos longos**. A exceção é para índices financeiros, tal como o Ibovespa, onde os dados do Yahoo Finance são bastante confiáveis uma vez que índices não sofrem os mesmos ajustamentos que ações individuais.

### 5.3 Pacote GetTDData

Arquivos com informações sobre preços e retornos de títulos emitidos pelo governo brasileiro podem ser baixados manualmente no site do Tesouro Nacional. O tesouro direto é um tipo especial de mercado onde pessoa física pode comprar e vender dívida pública. Os contratos de dívida vendidos na plataforma são bastante populares devido a atratividade das taxas de retorno e a alta liquidez oferecida ao investidor comum.

Pacote `GetTDData` importa os dados das planilhas em Excel do site do Tesouro Nacional e os organiza. O resultado é um `dataframe` com dados empilhados. Como exemplo, vamos baixar dados de um título prefixado do tipo LTN com vencimento em 2021-01-01. Esse é o tipo de contrato de dívida mais simples que o governo brasileiro emite, não pagando nenhum cupom<sup>3</sup> durante sua validade e, na data de vencimento, retorna 1.000 R\$ ao comprador. Para baixar os dados da internet, basta usar o código a seguir:

<sup>3</sup>O cupom é um pagamento intermediário pago periodicamente durante a validade do contrato financeiro.

```
library(GetTDData)

asset_codes <- 'LTN' # Identifier of assets
maturity <- '010121' # Maturity date as string (ddmmyy)

# download
my_flag <- download.TD.data(asset_codes = asset_codes)

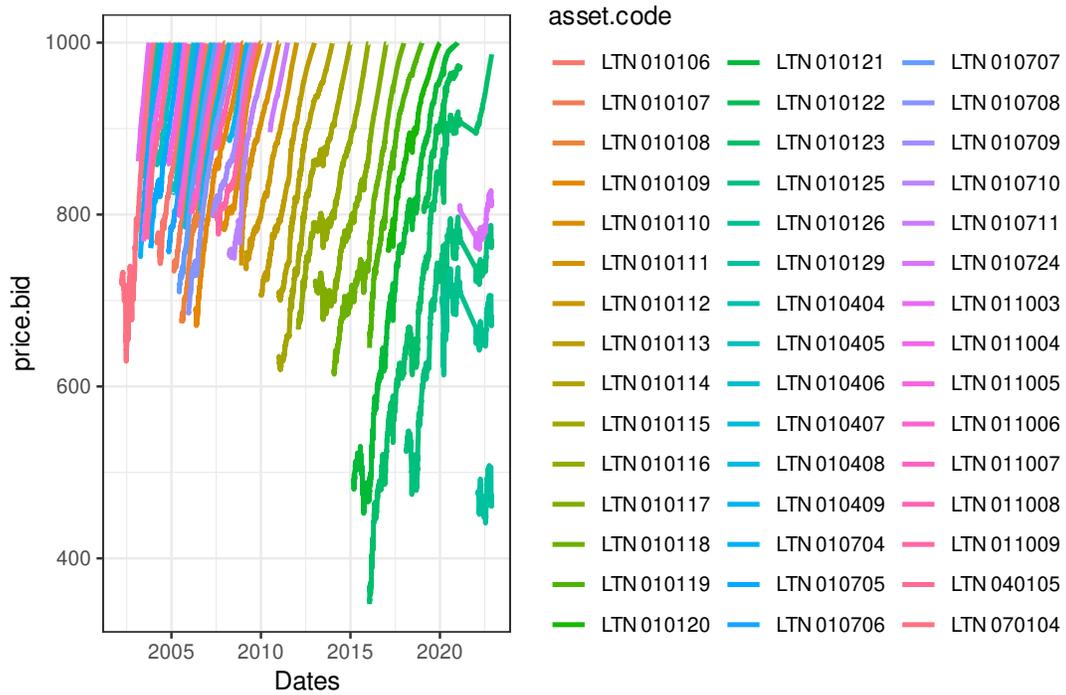
# read files
df_TD <- read.TD.files()
```

Vamos checar o conteúdo do dataframe:

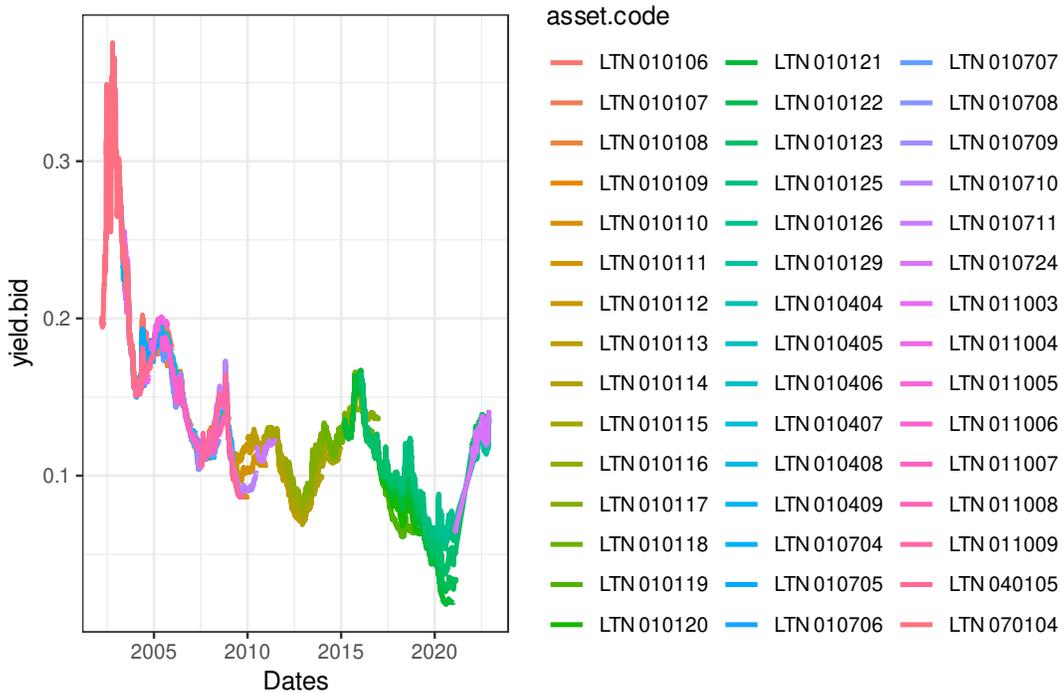
```
# check content
glimpse(df_TD)
```

```
R> Rows: 24,733
R> Columns: 5
R> $ ref.date <date> 2002-03-18, 2002-03-19, 2002-03-20, 20~
R> $ yield.bid <dbl> 0.1977, 0.1975, 0.1967, 0.2005, 0.1983,~
R> $ price.bid <dbl> 720.96, 721.70, 723.09, 719.48, 722.38,~
R> $ asset.code <chr> "LTN 070104", "LTN 070104", "LTN 070104~
R> $ matur.date <date> 2004-01-07, 2004-01-07, 2004-01-07, 20~
```

Temos informações sobre data de referência (`ref.date`), retorno contratado (`yield.bid`), preço do contrato na data (`price.bid`), nome do contrato (`asset.code`) e dia de maturidade (`matur.date`). No gráfico a seguir checamos os dados:



Como esperado de um título de dívida pré-fixado, os preços possuem uma tendência positiva ao longo do tempo, chegando ao valor esperado de 1000 R\$ no vencimento em 2021-01-01. Podemos também visualizar as mudanças do *yield* do título:



Os retornos do título tiveram forte queda ao longo dos anos. Este resultado é esperado pois o juros do mercado – taxa SELIC – caiu bastante nos últimos cinco anos.

As funções do `GetTDData` também funcionam com vários argumentos como `asset.codes` e `maturity`. Suponhamos que desejamos visualizar todos os preços de todos os prazos disponíveis para títulos do tipo LTN a partir de 2010. Tudo o que precisamos fazer é adicionar o valor `NULL` ao argumento `maturity` e filtrar as datas:

```
library(GetTDData)

asset_codes <- 'LTN' # Name of asset
maturity <- NULL    # = NULL, downloads all maturities

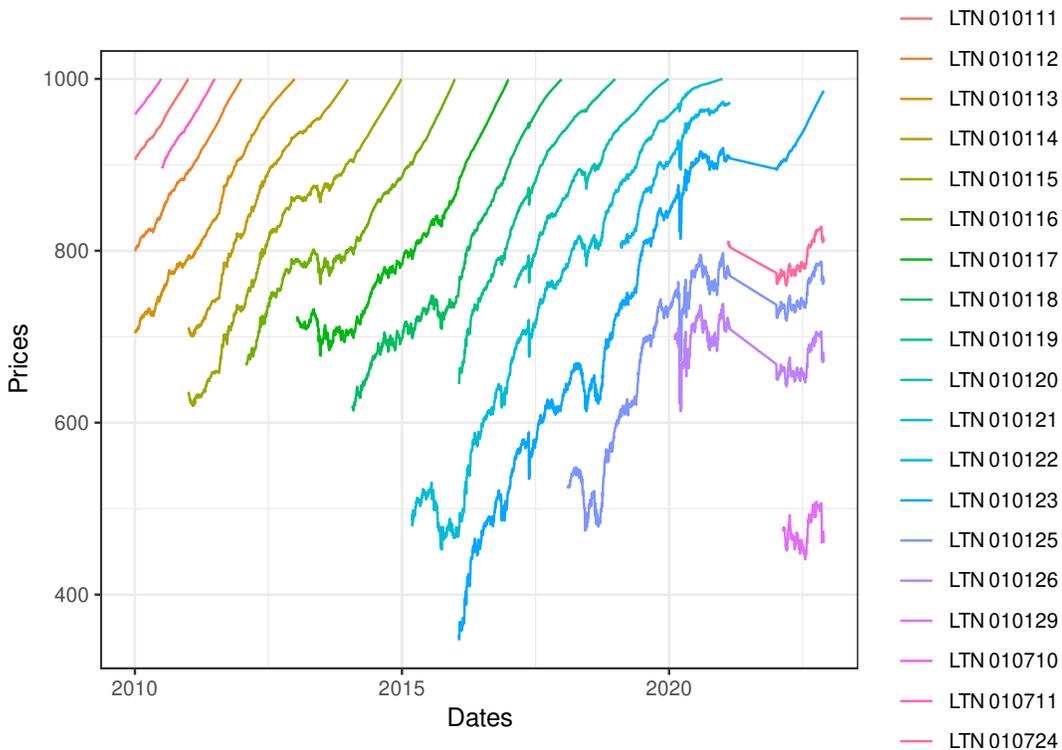
# download data
my_flag <- download.TD.data(asset.codes = asset_codes,
                           do.clean.up = F)

# reads data
df_TD <- read.TD.files()

# remove data prior to 2010
```

```
df_TD <- dplyr::filter(df_TD,
                      ref.date >= as.Date('2010-01-01'))
```

Após a importação das informações, plotamos os preços dos diferentes ativos:



Note como todos contratos do tipo LTN terminam com valor R\$ 1.000 em sua data de expiração e possuem uma dinâmica linear de crescimento de preço ao longo do tempo.

Outra funcionalidade do pacote `GetTDDData` é o acesso a curva de juros atual do sistema financeiro brasileiro diretamente do site da Anbima. Para isso, basta utilizar a função `get.yield.curve`:

```
library(GetTDDData)

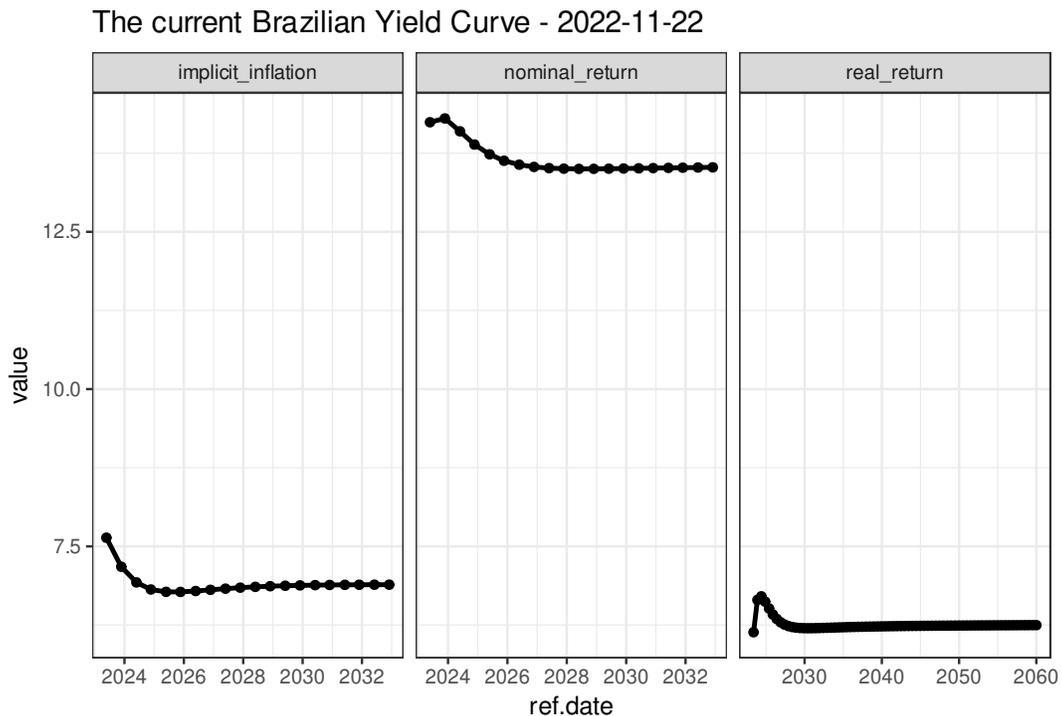
# get yield curve
df_yield <- get.yield.curve()

# check result
dplyr::glimpse(df_yield)
```

R> Rows: 114

```
R> Columns: 5
R> $ n.biz.days <dbl> 126, 252, 378, 504, 630, 756, 882, 10~
R> $ type <chr> "real_return", "real_return", "real_r~
R> $ value <dbl> 6.1355, 6.6490, 6.7057, 6.6214, 6.512~
R> $ ref.date <date> 2023-05-24, 2023-11-23, 2024-05-27, ~
R> $ current.date <date> 2022-11-22, 2022-11-22, 2022-11-22, ~
```

Os dados incluem a curva de juros nominal, juros real e da inflação. Para melhor visualizar as informações, vamos plotá-las em um gráfico:



A curva de juros é uma ferramenta utilizada no mercado financeiro com o propósito de representar graficamente a expectativa do mercado sobre juros futuro. Baseada nos preços dos títulos públicos, calcula-se e extrapola-se o juros implícito para cada período futuro. Uma curva ascendente, o formato esperado, indica que é mais caro (maior o juro) tomar dinheiro emprestado no longo prazo.

## 5.4 Pacote GetBCBData

O Banco Central Brasileiro (BCB) disponibiliza em seu Sistema de Séries Temporais (SGS) uma vasta quantidade de tabelas relativas a economia do Brasil. Mais importante, estas tabelas são atualizadas constantemente e o acesso é gratuito e sem necessidade de registro.

Como um exemplo, vamos usar o pacote para estudar a inadimplência de crédito no sistema financeiro Brasileiro. O primeiro passo no uso de `GetBCBData` é procurar o símbolo da série de interesse. Acessando o sistema de séries temporais do BCB, vemos que o código identificador para o percentual total de inadimplência no Brasil é 21082.

No código, basta indicar a série de interesse e o período de tempo desejado:

```
library(GetBCBData)
library(dplyr)

# set ids and dates
id_series <- c(perc_default = 21082)
first_date = '2010-01-01'

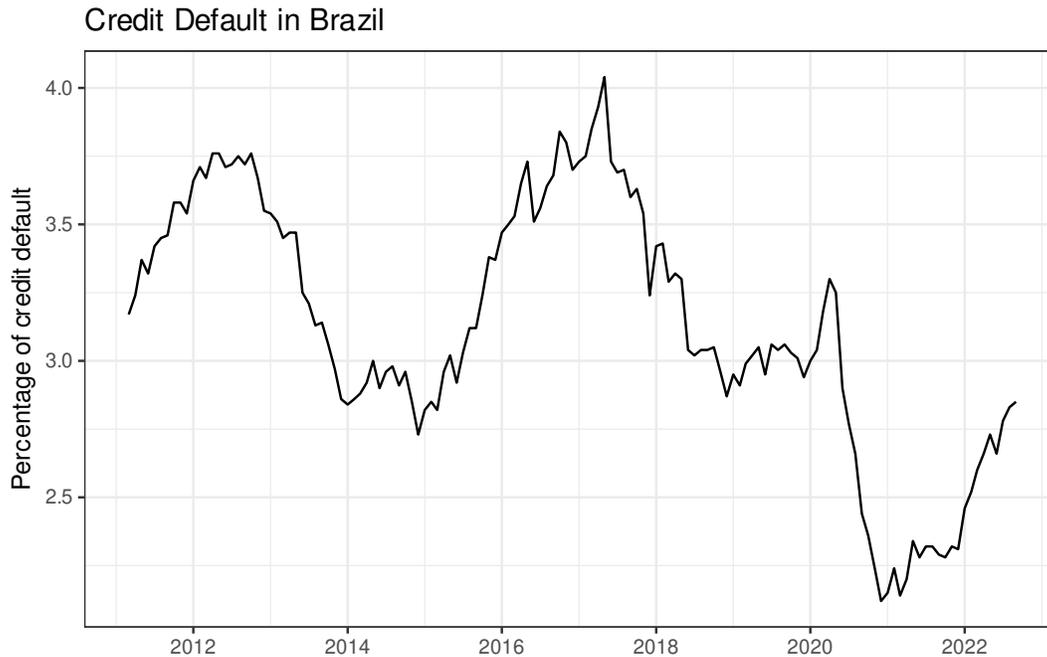
# get series from bcb
df_cred <- gcbcb_get_series(id = id_series,
                           first.date = first_date,
                           last.date = Sys.Date(),
                           use.memoise = FALSE)
```

```
R>
R> Fetching perc_default [21082] from BCB-SGS from Online API
R> Found 139 observations
```

```
# check it
glimpse(df_cred)
```

```
R> Rows: 139
R> Columns: 4
R> $ ref.date    <date> 2011-03-01, 2011-04-01, 2011-05-01, 2~
R> $ value      <dbl> 3.17, 3.24, 3.37, 3.32, 3.42, 3.45, 3.~
R> $ id.num     <dbl> 21082, 21082, 21082, 21082, 21082, 210~
R> $ series.name <chr> "perc_default", "perc_default", "perc_~
```

Note que indicamos o nome da coluna na própria definição da entrada `id`. Assim, coluna `series.name` toma o nome de `perc.default`. Esta configuração é importante pois irá diferenciar os dados no caso da importação de diversas séries diferentes. O gráfico apresentado a seguir mostra o valor da série no tempo:



Source: SGS - BCB (by GetBCBData)

Como podemos ver, a porcentagem de inadimplência aumentou a partir de 2015. Para ter uma idéia mais clara do problema, vamos incluir no gráfico a porcentagem para pessoa física e pessoa jurídica. Olhando novamente o sistema do BCB, vemos que o símbolos de interesse são 21083 e 21084, respectivamente. O próximo código baixa os dados das duas séries.

```
# set ids
id.series <- c(credit_default_people = 21083,
              credit_default_companies = 21084)
first.date = '2010-01-01'

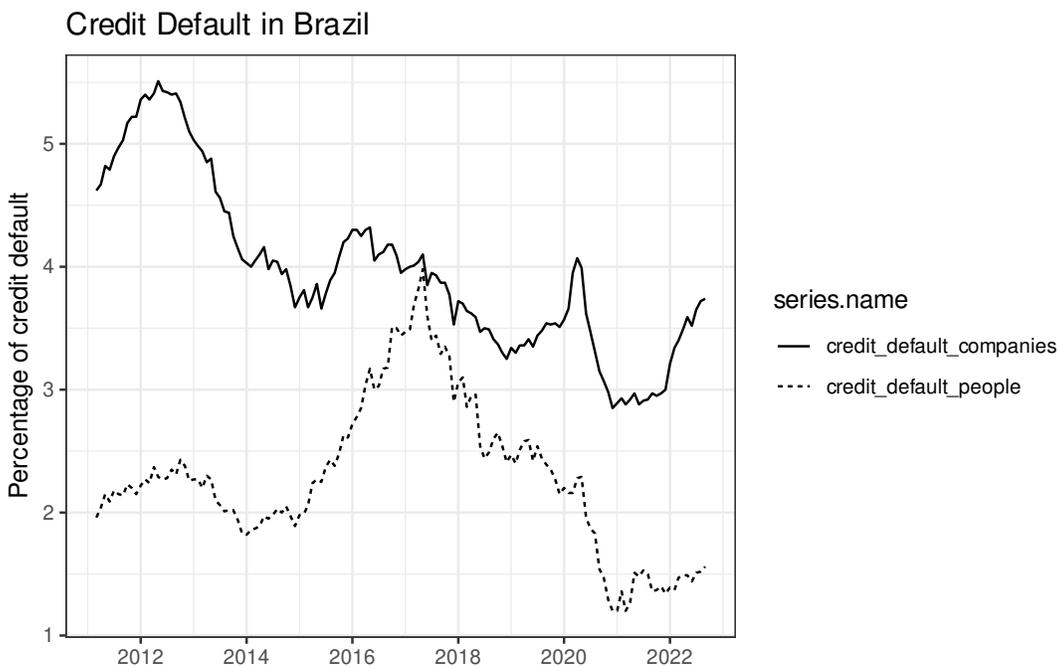
# get series from bcb
df_cred <- gcbcb_get_series(id = id.series,
                           first.date = first.date,
                           last.date = Sys.Date(),
                           use.memoise = FALSE)
```

```
R>
R> Fetching credit_default_people [21083] from BCB-SGS from Online API
R> Found 139 observations
R> Fetching credit_default_companies [21084] from BCB-SGS from Online API
R> Found 139 observations
```

```
# check output
glimpse(df_cred)
```

```
R> Rows: 278
R> Columns: 4
R> $ ref.date    <date> 2011-03-01, 2011-04-01, 2011-05-01, 2~
R> $ value      <dbl> 1.96, 2.04, 2.15, 2.09, 2.18, 2.15, 2.~
R> $ id.num     <dbl> 21083, 21083, 21083, 21083, 21083, 210~
R> $ series.name <chr> "credit_default_people", "credit_defau~
```

A diferença na saída do código anterior é que agora temos duas séries temporais empilhadas no mesmo `dataframe`. Partimos então para a visualização das séries



Source: SGS - BCB (by GetBCBData)

Como podemos ver, a inadimplência de crédito para pessoa física aumentou muito mais do que a para pessoa jurídica (empresas) nos últimos anos. Poderíamos, facilmente, integrar o código anterior para uma análise mais completa dos dados em algum problema de pesquisa.



O sistema BCB-SGS é local obrigatório para qualquer economista sério. A quantidade e variedade de dados é imensa. Podes usar os dados do sistema para automatizar qualquer tipo de relatório econômico.

## 5.5 Pacote GetDFPData2

Pacote `GetDFPData2` (Perlin and Kirch, 2022a) é uma evolução do pacote `GetDFPData` (Perlin, 2021) e fornece uma interface aberta para todas as demonstrações financeiras distribuídas pela B3 e pela CVM nos sistemas DFP (dados anuais) e ITR (dados trimestrais). Ele não só faz o *download* dos dados, mas também ajusta à inflação e torna as tabelas prontas para pesquisa dentro de um formato tabular. Os diferenciais do pacote em relação a outros distribuidores de dados comerciais são: livre acesso, facilidade para baixar dados em larga escala e a variedade de dados disponíveis.



Uma versão web de `GetDFPData2` foi desenvolvida e publicada na internet como um aplicativo *shiny* em <http://www.msperlin.com/shiny/GetDFPData/>. Esse fornece uma interface gráfica direta e simples para as principais funcionalidade do pacote. Usuários podem selecionar as empresas disponíveis, o intervalo de datas e baixar os dados como uma planilha do Excel ou um arquivo compactado com vários arquivos *csv*. Saiba também que dados históricos completos e atualizados a partir de 2010 do DFP e ITR estão disponibilizados na seção Data do meu site pessoal.

O ponto de partida no uso de `GetDFPData2` é baixar informações atuais sobre empresas disponíveis. O acesso a tabela é possível com a função `get_info_companies`:

```
library(GetDFPData2)

# get info for companies in B3
df_info <- get_info_companies()

R> Fetching info on B3 companies

R> Found cache file. Loading data..

R> Got 2559 lines for 2425 companies [Actives = 701 Inactives = 1736]

# check it
names(df_info)

R> [1] "CNPJ" "DENOM_SOCIAL"
R> [3] "DENOM_COMERC" "DT_REG"
R> [5] "DT_CONST" "DT_CANCEL"
R> [7] "MOTIVO_CANCEL" "SIT_REG"
R> [9] "DT_INI_SIT" "CD_CVM"
```

```

R> [11] "SETOR_ATIV"          "TP_MERC"
R> [13] "CATEG_REG"          "DT_INI_CATEG"
R> [15] "SIT_EMISSOR"        "DT_INI_SIT_EMISSOR"
R> [17] "CONTROLE_ACIONARIO" "TP_ENDER"
R> [19] "LOGRADOURO"         "COMPL"
R> [21] "BAIRRO"             "MUN"
R> [23] "UF"                 "PAIS"
R> [25] "CEP"                "DDD_TEL"
R> [27] "TEL"                "DDD_FAX"
R> [29] "FAX"                "EMAIL"
R> [31] "TP_RESP"            "RESP"
R> [33] "DT_INI_RESP"        "LOGRADOURO_RESP"
R> [35] "COMPL_RESP"         "BAIRRO_RESP"
R> [37] "MUN_RESP"           "UF_RESP"
R> [39] "PAIS_RESP"          "CEP_RESP"
R> [41] "DDD_TEL_RESP"       "TEL_RESP"
R> [43] "DDD_FAX_RESP"       "FAX_RESP"
R> [45] "EMAIL_RESP"         "CNPJ_AUDITOR"
R> [47] "AUDITOR"

```

Essa tabela disponibiliza os identificadores numéricos das empresas, setores de atividades, atual segmento de governança, *tickers* negociados na bolsa e situação atual (ativa ou não). O número atual de empresas ativas e inativas, a partir de 2022-11-23, está disponível na coluna `SIT_REG`. Observa-se 786 empresas ativas e 1771 canceladas. Essa é uma excelente fonte de informação para um estudo exploratório. Pode-se facilmente filtrar empresas para datas, setores, *tickers* ou segmentos de governança corporativa.

Toda empresa no banco de dados é identificada pelo seu número único da CVM. Função `search_company` permite que o usuário procure o identificador de uma empresa através de seu nome. Dado um texto de entrada – o nome da empresa –, a função procurará uma correspondência parcial com os nomes de todas as empresas disponíveis no banco de dados. Em seu uso, caracteres latinos e maiúsculas e minúsculas são ignorados. Vamos encontrar o nome oficial nome da Grendene, uma das maiores empresas do Brasil. Para isso, basta usar o comando `search_company('grendene')`.

```
df_search <- search_company('grendene')
```

```
print(df_search)
```

```

R> # A tibble: 1 x 47
R>   CNPJ      DENOM~1 DENOM~2 DT_REG      DT_CONST  DT_CANCEL
R>   <chr>    <chr>    <chr>  <date>    <date>    <date>
R> 1 89.850.3~ GRENDE~ GRENDE~ 2004-10-26 1971-02-25 NA

```

```
R> # ... with 41 more variables: MOTIVO_CANCEL <chr>,
R> #   SIT_REG <chr>, DT_INI_SIT <date>, CD_CVM <dbl>,
R> #   SETOR_ATIV <chr>, TP_MERC <chr>, CATEG_REG <chr>,
R> #   DT_INI_CATEG <date>, SIT_EMISSOR <chr>,
R> #   DT_INI_SIT_EMISSOR <date>, CONTROLE_ACIONARIO <chr>,
R> #   TP_ENDER <chr>, LOGRADOURO <chr>, COMPL <chr>,
R> #   BAIRRO <chr>, MUN <chr>, UF <chr>, PAIS <chr>, ...
```

Vemos que existe um registro para a Grendene: “GRENDENE SA”, com código identificador equivalente a 19615.

Com o identificador da empresa disponível, usamos a função principal do pacote, `get_dfp_data`, para baixar os dados. Definimos o nome oficial da empresa como entrada `companies_cvm_codes` e o período de tempo como entradas `first_year` e `last_year`.

```
library(GetDFPData2)
library(dplyr)

# set options
id_companies <- 19615
first_year <- 2017
last_year <- 2018

# download data
l_dfp <- get_dfp_data(companies_cvm_codes = id_companies,
                      type_docs = '*', # get all docs
                      type_format = 'con', # consolidated
                      first_year = first_year,
                      last_year = last_year)
```

As mensagens de `GetDFPData2::get_dfp_data` relatam os estágios do processo, desde a aquisição de dados da tabela de referência ao download e leitura dos arquivos da B3. Observe que os arquivos de três sistemas são acessados: DFP (*Demonstrativos Financeiros Padronizados*), FRE (*Formulário de Referência*) e FCA (*Formulário Cadastral*). Observe também o uso de um sistema de cache, o qual acelera significativamente o uso do software ao salvar localmente as informações importadas.

Explicando as demais entradas da função `GetDFPData2::get_dfp_data`:

**companies\_cvm\_codes** Código numérico das empresas (encontrado via `GetDFPData2::search_company('ambev')`)

**type\_docs** Símbolo do tipo de documento financeiro a ser retornado. Definições: '\*' = retorna todos documentos, 'BPA' = Ativo, 'BPP' = passivo, 'DRE' = demonstrativo de resultados do exercício, 'DFC\_MD' = fluxo de caixa pelo

metodo direto, 'DFC\_MI' = fluxo de caixa pelo metodo indireto, 'DMPL' = mutacoes do patrimonio liquido, 'DVA' = demonstrativo de valor agregado.

**type\_format** Tipo de formato dos documentos: consolidado ('con') ou individual ('ind'). Como regra, dê preferência ao tipo consolidado, o qual incluirá dados completos de subsidiárias.

**first\_year** Primeiro ano para os dados

**last\_year** Último ano para os dados

O objeto resultante de `get_dfp_data` é uma lista com diversas tabelas. Vamos dar uma olhada no conteúdo de `l_dfp` ao buscar os nomes dos itens da lista, limitando o número de caracteres:

```
stringr::str_sub(names(l_dfp), 1, 40)
```

```
R> [1] "DF Consolidado - Balanço Patrimonial Ati"
R> [2] "DF Consolidado - Balanço Patrimonial Pas"
R> [3] "DF Consolidado - Demonstração das Mutaçõ"
R> [4] "DF Consolidado - Demonstração de Valor A"
R> [5] "DF Consolidado - Demonstração do Fluxo d"
R> [6] "DF Consolidado - Demonstração do Resulta"
```

Como podemos ver, os dados retornados são vastos. Cada item da lista em `l_dfp` é um tabela indexada ao tempo. A explicação de cada coluna não cabe aqui mas, para fins de exemplo, vamos dar uma olhada no balanço patrimonial da empresa, disponível em `l_dfp$"DF Consolidado - Balanço Patrimonial Ativo"`:

```
# save assets in df
fr_assets <- l_dfp$"DF Consolidado - Balanço Patrimonial Ativo"

# check it
dplyr::glimpse(fr_assets)
```

```
R> Rows: 122
R> Columns: 16
R> $ CNPJ_CIA      <chr> "89.850.341/0001-60", "89.850.341/000~
R> $ CD_CVM       <dbl> 19615, 19615, 19615, 19615, 19615, 19~
R> $ DT_REFER     <date> 2017-12-31, 2017-12-31, 2017-12-31, ~
R> $ DT_INI_EXERC <date> NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
R> $ DT_FIM_EXERC <date> 2017-12-31, 2017-12-31, 2017-12-31, ~
R> $ DENOM_CIA    <chr> "GRENDENE S.A.", "GRENDENE S.A.", "GR~
R> $ VERSAO       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
R> $ GRUPO_DFP    <chr> "DF Consolidado - Balanço Patrimonial~
R> $ MOEDA        <chr> "REAL", "REAL", "REAL", "REAL", "REAL~
R> $ ESCALA_MOEDA <chr> "MIL", "MIL", "MIL", "MIL", "MIL", "M~
R> $ ORDEM_EXERC  <chr> "ÚLTIMO", "ÚLTIMO", "ÚLTIMO", "ÚLTIMO~
```

```
R> $ CD_CONTA      <chr> "1", "1.01", "1.01.01", "1.01.02", "1~
R> $ DS_CONTA      <chr> "Ativo Total", "Ativo Circulante", "C~
R> $ VL_CONTA      <dbl> 3576008, 2846997, 30119, 1537477, 836~
R> $ COLUNA_DF     <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
R> $ source_file   <chr> "dfp_cia_aberta_BPA_con_2017.csv", "d~
```

A exportação dos dados para o Excel também é fácil, basta usar função `GetDFPData2::export_xlsx`:

```
temp_xlsx <- tempfile(fileext = '.xlsx')

export_xlsx(l_dfp = l_dfp, f_xlsx = temp_xlsx)
```

O arquivo Excel resultante conterà cada tabela de `l_dfp` em uma aba diferente da planilha, com uma truncagem nos nomes. Podemos checar o resultado com função `readxl::excel_sheets`:

```
readxl::excel_sheets(temp_xlsx)

R> [1] "DF Consolidado - Balanço Pat"
R> [2] "DF Consolidado - Balanço Pat_1"
R> [3] "DF Consolidado - Demonstraçã"
R> [4] "DF Consolidado - Demonstraçã_1"
R> [5] "DF Consolidado - Demonstraçã_2"
R> [6] "DF Consolidado - Demonstraçã_3"
```

## 5.6 Pacote GetFREData

O pacote `GetFREData` importa dados do sistema FRE – Formulário de Referência – da bolsa Brasileira, incluindo eventos e informações corporativas tal como composição do conselho e diretoria, remuneração dos conselhos, entre outras.

A estrutura de uso e a saída das funções de `GetFREData` são muito semelhante as do pacote `GetDFPData2`. Veja a seguir um exemplo de uso.

```
library(GetFREData)

# set options
id_companies <- 23264
first_year <- 2017
last_year <- 2018

# download data
l_fre <- get_fre_data(companies_cvm_codes = id_companies,
                     first_year = first_year,
```

```
last_year = last_year)
```

Note que o tempo de execução de `get_fre_data` é significativo. Isto deve-se ao *download* e leitura dos arquivos do sistema FRE direto da bolsa. Cada tabela do FRE é importada na lista de saída:

```
names(l_fre)
```

```
R> [1] "df_stockholders"
R> [2] "df_capital"
R> [3] "df_stock_values"
R> [4] "df_mkt_value"
R> [5] "df_increase_capital"
R> [6] "df_capital_reduction"
R> [7] "df_compensation"
R> [8] "df_compensation_summary"
R> [9] "df_transactions_related"
R> [10] "df_other_events"
R> [11] "df_stock_repurchases"
R> [12] "df_debt_composition"
R> [13] "df_board_composition"
R> [14] "df_committee_composition"
R> [15] "df_family_relations"
R> [16] "df_family_related_companies"
R> [17] "df_auditing"
R> [18] "df_responsible_docs"
R> [19] "df_stocks_details"
R> [20] "df_dividends_details"
R> [21] "df_intangible_details"
```

Por exemplo, vamos verificar conteúdo da tabela `df_board_composition`, a qual contém informações sobre os componentes dos conselhos das empresas:

```
glimpse(l_fre$df_board_composition)
```

```
R> Rows: 59
R> Columns: 22
R> $ CNPJ_CIA <chr> "07.526.557/0001-00", "07~
R> $ DENOM_CIA <chr> "AMBEV S.A.", "AMBEV S.A.~
R> $ DT_REFER <date> 2017-01-01, 2017-01-01, ~
R> $ CD_CVM <dbl> 23264, 23264, 23264, 2326~
R> $ ID_DOC <dbl> 74969, 74969, 74969, 7496~
R> $ VERSAO <dbl> 10, 10, 10, 10, 10, 10, 1~
R> $ person.name <chr> "Paula Nogueira Lindenber~"
```

```

R> $ person.cpf <dbl> 26712117836, 25661215835, ~
R> $ person.profession <chr> "Administradora", "Engenh~
R> $ person.cv <chr> "Nos últimos 5 anos, ocup~
R> $ person.dob <date> NA, NA, NA, NA, NA, NA, ~
R> $ code.type.board <chr> "1", "1", "1", "1", "1", ~
R> $ desc.type.board <chr> "Director", "Director", "~
R> $ desc.type.board2 <chr> "Diretora de Marketing", ~
R> $ code.type.job <chr> "19", "19", "19", "19", "~
R> $ desc.job <chr> "Não aplicável, uma vez q~
R> $ date.election <date> 2016-05-11, 2016-05-11, ~
R> $ date.effective <date> 2016-05-11, 2016-05-11, ~
R> $ mandate.duration <chr> "11/05/2019", "30/06/2018~
R> $ ellected.by.controller <lgl> TRUE, TRUE, TRUE, TRUE, T~
R> $ qtd.consecutive.mandates <dbl> 2, 2, 2, 2, 2, 1, 2, 1, 1~
R> $ percentage.participation <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0~

```

Como podemos ver, para um pesquisador de finanças corporativas, o sistema FRE oferece uma série de informações interessantes. Discutir o conteúdo de cada tabela, porém, vai muito além do propósito dessa seção. Aos interessados, mais detalhes sobre as tabelas do FRE estão disponíveis em Perlin et al. (2018).



Note que a importação dos dados do FRE inclui uma versão dos arquivos. Toda vez que uma empresa modifica as informações oficiais no sistema da B3, uma nova versão do FRE é criada. Devido a isso, é bastante comum que os dados de um ano para uma empresa possuam diferentes versões. Para resolver este problema, o código do `GetFREData`, por *default*, importa a versão mais antiga para cada ano. Caso o usuário queira mudar, basta utilizar a entrada `fre_to_read`.

## 5.7 Outros Pacotes

Nas seções anteriores destacamos os principais pacotes gratuitos para aquisição de dados financeiros e econômicos no Brasil. Muitos desses foram escritos pelo próprio autor do livro e representam uma pequena parcela da totalidade. Não seria justo ignorar o trabalho de outros autores. Assim, reporto abaixo uma seleção de pacotes que vale a pena conhecer:

### 5.7.1 Pacotes de Acesso Gratuito

**BETS** (Ferreira et al., 2018) Pacote construído e mantido pela equipe da FGV. Permite o acesso aos dados do BCB (Banco Central do Brasil) e IBGE (Instituto Brasileiro de Geografia e Estatística). Também inclui ferramentas para

a administração, análise e manipulação dos dados em relatórios técnicos.

**simfinapi** (Gomolka, 2021) Pacote para acesso ao projeto simfin, incluindo dados financeiros de diversas empresas internacionais. O acesso livre é restrito a um número de chamadas diárias.

### 5.7.2 Pacotes Comerciais

**Rblpapi** (Armstrong et al., 2022) Permite acesso aos dados da Bloomberg, sendo necessário uma conta comercial.

**IBrokers** (Ryan and Ulrich, 2022a) API para o acesso aos dados da Interactive Brokers. Também é necessário uma conta comercial.

No CRAN você encontrará muitos outros. A interface para fontes de dados comerciais também é possível. Várias empresas fornecem APIs para facilitar o envio de dados aos seus clientes. Se a empresa de fornecimento de dados que você usa no trabalho não for apresentada aqui, a lista de pacotes CRAN pode ajudá-lo a encontrar uma alternativa viável.

## 5.8 Acessando Dados de Páginas na Internet (*Webscraping*)

Os pacotes destacados anteriormente são muito úteis pois facilitam a importação de dados específicos diretamente da internet. Em muitos casos, porém, os dados de interesse não estão disponíveis via API formal, mas sim em uma página na internet - geralmente no formato de uma tabela. O processo de extrair informações de páginas da internet chama-se *webscraping* (raspagem de dados). Dependendo da estrutura e da tecnologia da página da *web* acessada, importar essas informações diretamente para o R pode ser um procedimento trivial – mas também pode se tornar um processo extremamente trabalhoso. Como um exemplo, a seguir vamos raspar dados do Wikipedia sobre a composição do índice SP500.

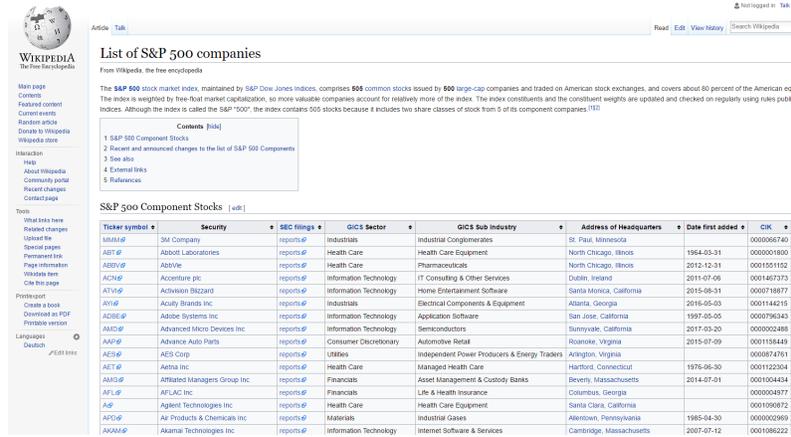
### 5.8.1 Raspando Dados do Wikipedia

Em seu site, a Wikipedia oferece uma seção<sup>4</sup> com os componentes do Índice SP500. Essas informações são apresentadas em um formato tabular, Figura 5.1.

As informações desta página são constantemente atualizadas, e podemos utilizá-las para importar informações sobre as ações pertencentes ao índice SP500. Antes de nos aprofundarmos no código R, precisamos entender como uma página da web funciona. Resumidamente, uma página da web nada mais é do que uma árvore com nódulos, representada por um código HTML (*Hypertext Markup Language*) extenso interpretado pelo seu navegador. Um valor numérico ou texto apresentado no site

<sup>4</sup>[https://en.wikipedia.org/wiki/List\\_of\\_S%26P\\_500\\_companies](https://en.wikipedia.org/wiki/List_of_S%26P_500_companies)

## 5.8. ACESSANDO DADOS DE PÁGINAS NA INTERNET (WEBSCRAPING)145



The screenshot shows the Wikipedia page for 'List of S&P 500 companies'. The page content includes a table of S&P 500 Component Stocks with columns for Ticker symbol, Security, SIC filings, GICS Sector, GICS Sub Industry, Address of Headquarters, Date first added, and CIK. The table lists various companies such as 3M Company, Abbott Laboratories, AbbVie, Accenture plc, Activision Blizzard, Acuity Brands Inc, Adobe Systems Inc, Advanced Micro Devices Inc, Advance Auto Parts, AES Corp, Aetna Inc, Affiliated Managers Group Inc, AFLAC Inc, Agilent Technologies Inc, Air Products & Chemicals Inc, and Akamai Technologies Inc.

Ticker symbol	Security	SIC filings	GICS Sector	GICS Sub Industry	Address of Headquarters	Date first added	CIK
MMM	3M Company	reports	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1964-03-31	000066740
ABT	Abbott Laboratories	reports	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	000001800
ABBV	AbbVie	reports	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	000105152
ACN	Accenture plc	reports	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	0001467373
ATVI	Activision Blizzard	reports	Information Technology	Home Entertainment Software	Santa Monica, California	2015-08-31	0000718977
ATY	Acuity Brands Inc	reports	Industrials	Electrical Components & Equipment	Atlanta, Georgia	2016-05-03	0001144215
ADBE	Adobe Systems Inc	reports	Information Technology	Application Software	San Jose, California	1997-05-05	0000796243
AMD	Advanced Micro Devices Inc	reports	Information Technology	Semiconductors	Sunnyvale, California	2017-03-20	0000020488
AAP	Advance Auto Parts	reports	Consumer Discretionary	Automotive Retail	Roanoke, Virginia	2015-07-09	0001158449
AES	AES Corp	reports	Utilities	Independent Power Producers & Energy Traders	Arlington, Virginia	1990	0000974701
AET	Aetna Inc	reports	Health Care	Managed Health Care	Hartford, Connecticut	1976-06-30	000122204
AMG	Affiliated Managers Group Inc	reports	Financials	Asset Management & Custody Banks	Beverly, Massachusetts	2014-07-01	0001004344
AFL	AFLAC Inc	reports	Financials	Life & Health Insurance	Columbus, Georgia	1990	000004377
ADI	Agilent Technologies Inc	reports	Health Care	Health Care Equipment	Santa Clara, California	1990	0001050872
APC	Air Products & Chemicals Inc	reports	Materials	Industrial Gases	Allentown, Pennsylvania	1985-04-30	0000022999
AKAM	Akamai Technologies Inc	reports	Information Technology	Internet Software & Services	Cambridge, Massachusetts	2007-07-12	000106222

Figura 5.1: Imagem da página do Wikipedia

geralmente pode ser encontrado dentro do próprio código. Este código tem uma estrutura particular em forma de árvore com ramificações, classes, nomes e identificadores. Além disso, cada elemento de uma página da web possui um endereço, denominado *xpath*. Nos navegadores Chrome e Firefox, você pode ver o código HTML de uma página da web usando o mouse. Para isto, basta clicar com o botão direito em qualquer parte da página e selecionar *View Page Source* (ou “Ver Código Fonte”).

A primeira etapa do processo de raspagem de dados é descobrir a localização das informações de que você precisa. No navegador Chrome, você pode fazer isso clicando com o botão direito no local específico do número/texto no site e selecionando *inspect*. Isso abrirá uma janela extra no navegador a direita. Depois de fazer isso, clique com o botão direito na seleção e escolha *copy* e *copy xpath*. Na Figura 5.2, vemos um espelho do que você deve estar vendo em seu navegador.

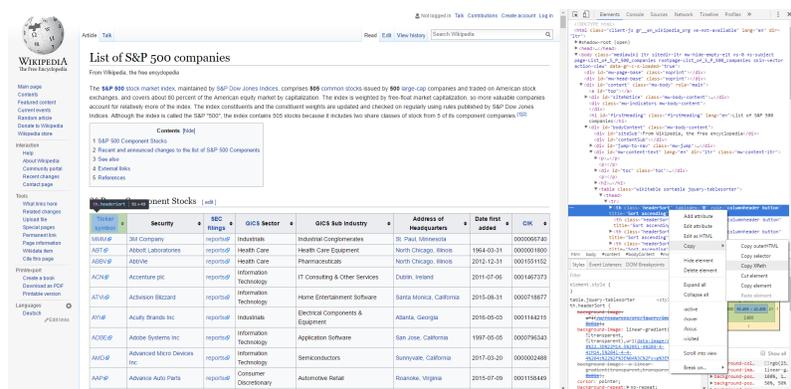


Figura 5.2: Encontrando o xpath da tabela

Aqui, o texto copiado é:

```
'//*[@id="mw-content-text"]/table[1]/thead/tr/th[2]'
```

Este é o endereço do cabeçalho da tabela. Para todo o conteúdo da tabela, incluindo cabeçalho, linhas e colunas, precisamos definir um nível superior da árvore. Isso é equivalente ao endereço `//*[@id = " mw-content-text"]/table[1]`.

Agora que temos a localização do que queremos, vamos carregar o pacote `rvest` (Wickham, 2022a) e usar as funções `read_html`, `html_nodes` e `html_table` para importar a tabela desejada para o R:

```
library(rvest)

# set url and xpath
my_url <- 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
my_xpath <- '//*[@id="mw-content-text"]/div/table[1]'

# get nodes from html
out_nodes <- html_nodes(read_html(my_url),
                        xpath = my_xpath)

# get table from nodes (each element in
# list is a table)
df_SP500_comp <- html_table(out_nodes)

# isolate it
df_SP500_comp <- df_SP500_comp[[1]]

# change column names (remove space)
names(df_SP500_comp) <- make.names(names(df_SP500_comp))

# print it
glimpse(df_SP500_comp)
```

```
R> Rows: 503
R> Columns: 9
R> $ Symbol <chr> "MMM", "AOS", "ABT", "ABBV", ~
R> $ Security <chr> "3M", "A. O. Smith", "Abbott~
R> $ SEC.filings <chr> "reports", "reports", "repor~
R> $ GICS.Sector <chr> "Industrials", "Industrials"~
R> $ GICS.Sub.Industry <chr> "Industrial Conglomerates", ~
R> $ Headquarters.Location <chr> "Saint Paul, Minnesota", "Mi~
R> $ Date.first.added <chr> "1976-08-09", "2017-07-26", ~
```

```
R> $ CIK           <int> 66740, 91142, 1800, 1551152,~
R> $ Founded      <chr> "1902", "1916", "1888", "201~
```

O objeto `df_SP500_comp` contém um espelho dos dados do site da Wikipédia. Os nomes das colunas requerem algum trabalho de limpeza, mas o principal está ali. Observe como a saída é semelhante aos dados da função `BatchGetSymbols::GetSP500Stocks`. A razão é simples, ambas buscaram a informação na mesma origem. A diferença é que função `GetSP500Stocks` vai um passo além, limpando os dados importados.

## 5.9 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Using the `yfR` package, download daily data for ticker MMM from *Yahoo Finance* for the period between 2019-01-01 and 2022-05-24. What is the lowest **unadjusted closing price (column price\_close)** in the analyzed period?

- a) 43.08903
- b) 65.59444
- c) 88.09984
- d) 20.58363
- e) 117.87

02 - If you have not already done so, create a profile on the Quandl website<sup>5</sup> and download the arabica coffee price data (id = CEPEA/COFFEE\_A) in the CEPEA database (Center for Advanced Studies in Applied Economics) between 2010-01-01 and 2020-12-28. What is the maximum price found?

- a) \$194.03
- b) \$50.30
- c) \$265.90
- d) \$122.16
- e) \$349.39

03 - Utilize pacote `GetBCBData` para baixar dados do IPCA mensal entre 2010-01-01 e 2020-12-31. Para os dados importados, qual é a data com o maior valor mensal de inflação?

- a) 2010-11-01
- b) 2010-10-01
- c) 2011-05-01
- d) 2012-04-01
- e) 2020-12-01

---

<sup>5</sup><https://www.quandl.com/>

04 - Use function `simfinapi::sfa_get_entities` to import data about all available companies in Simfin. How many companies do you find?

- a) 4945
- b) 3229
- c) 4311
- d) 5579
- e) 3677

05 - With package `simfinapi`, download the PnL (profit and loss) statement for FY (final year) data for `id = 998466` for year 2015. What is the latest “Net Income” of the company?

- a) \$404.96
- b) \$534.36
- c) \$469.66
- d) \$599.07
- e) \$349.39

06 - Using again the `yfR` package, download data between 2019-01-01 and 2021-10-12 for the following tickers:

- AAPL: Apple Inc
- BAC: Bank of America Corporation
- GE: General Electric Company
- TSLA: Tesla, Inc.
- SNAP: Snap Inc.

Using the **adjusted closing price** column, what company provided higher return to the stock holder during the analyzed period?

Tip: this is an advanced exercise that will require some coding. To solve it, check out function `split` to split the dataframe of price data and `lapply` to map a function to each dataframe.

- a) TSLA
- b) AAPL
- c) BAC
- d) SNAP
- e) GE

07 - Using package `GetDFPData2`, download the main table with all available companies. What is the number of active companies currently available?

- a) 1050
- b) 888
- c) 1372

- d) 786
- e) 1211

08 - Using package `GetDFPData2`, download net income data (consolidated) for all companies in the year of 2018. What is the sum of profit/loss (in thousands, code 3.11 in DFP) for all companies?

- a) \$382,399,640
- b) \$269,132,608
- c) \$56,155,004
- d) \$127,147,539
- e) \$198,140,073

09 - Using package `GetTDDData`, download all available data for debt contract of type LTN (*pré-fixados*).

What is the maximum yield (column *yield.bid*) found for for contract LTN 010409?

- a) 5.8%
- b) 20.0%
- c) 35.2%
- d) 15.2%
- e) 10.2%



# Capítulo 6

## Dataframes e outros Objetos

**No R, tudo é um objeto.** Sempre que chamamos uma função, tal como nos exemplos dos capítulos anteriores, ela nos retorna um objeto. Cada tipo ou classe de objeto terá uma série de propriedades diferentes. Por exemplo, um `dataframe` pode ser incrementado com novas colunas ou linhas. Uma coluna numérica de um `dataframe` pode interagir com outros valores numéricos através de operações de multiplicação, divisão e soma. Para colunas com textos, porém, tal propriedade não é válida, uma vez que não faz sentido somar um valor numérico a um texto ou dividir um texto por outro. Entretanto, a classe de texto tem outras propriedades, como a que permite procurar uma determinada sequência textual dentro de um texto maior, a manipulação de partes do texto e a substituição de caracteres específicos, dentre tantas outras possibilidades. **Um dos aspectos mais importantes no trabalho com o R é o aprendizado das classes de objetos e as suas funcionalidades.**

As classes básicas de objetos no R inclui valores numéricos, caracteres (texto), fatores, datas, entre vários outros casos. Na prática, porém, as classes básicas são armazenadas em estruturas de dados mais complexas, tal como `dataframes` e listas. Isso organiza e facilita o trabalho. Imagine realizar um estudo sobre as 63 ações que compõem o índice Ibovespa, onde a base de dados é composta por preços e volumes negociados ao longo de um ano. Caso fôssemos criar um vetor numérico de preços e de volumes para cada ação, teríamos uma quantidade de 126 objetos para lidar no nosso *environment*. Apesar de ser possível trabalhar dessa forma, o código resultante seria desorganizado, difícil de entender e passível de uma série de erros.

Uma maneira mais simples de organizar os nossos dados é criar um objeto com o nome `my_data` e alocar todos os preços e volumes ali. Todas as informações necessárias para executar a pesquisa estariam nesse objeto, facilitando a importação e exportação dos dados. Esses objetos que armazenam outros objetos de classe básica

constituem a classe de estrutura de dados. Nessa classificação, estão incluídas listas (`list`), matrizes (`matrix`) e tabelas (`dataframes`).

## 6.1 Dataframes

Traduzindo para o português, `dataframe` significa “estrutura ou organização de dados”. Grosso modo, um objeto da classe `dataframe` nada mais é do que uma tabela com linhas e colunas. Sem dúvida, **o `dataframe` é o principal objeto utilizado no trabalho com o R e o mais importante de se estudar**. Dados externos são, grande maioria dos casos, importados para o R no formato de tabelas. É na manipulação desses que gastará maior parte do tempo realizando a sua análise. Internamente, um `dataframe` é um tipo especial de lista, onde cada coluna é um vetor atômico com o mesmo número de elementos, porém com sua própria classe. Podemos organizar em um `dataframe` dados de texto juntamente com números, por exemplo.

Note que o formato tabular força a sincronização dos dados no sentido de linhas, isto é, cada caso de cada variável deve ser pareado com casos de outras variáveis. Apesar de simples, esse tipo de estruturação de dados é intuitiva e pode acomodar uma variedade de informações. Cada acréscimo de dados (informações) incrementa as linhas e cada novo tipo de informação incrementa as colunas da tabela.

Um dos pontos positivos na utilização do `dataframe` para a acomodação de dados é que funções de diferentes pacotes irão funcionar a partir dessa classe de objetos. Por exemplo, o pacote de manipulação de dados `dplyr`, assim como o pacote de criação de figuras `ggplot2`, funcionam a partir de um `dataframe`. Esse objeto, portanto, está no centro de uma série de funcionalidades do R e, sem dúvida, é uma classe de objeto extremamente importante para aprender a utilizar corretamente.

O objeto `dataframe` é uma das classes nativas do R e vem implementado no pacote `base`. Entretanto, o universo `tidyverse` oferece sua própria versão de um `dataframe`, chamada `tibble`, a qual é utilizada sistematicamente em todos pacotes do `tidyverse`. A conversão de um `dataframe` para `tibble` é interna e automática. O `tibble` possui propriedades mais flexíveis que `dataframes` nativos, facilitando de forma significativa o seu uso. Seguindo a nossa preferência para o `tidyverse`, a partir de agora iremos utilizar `tibbles` como representantes de `dataframes`.

### 6.1.1 Criando dataframes

A criação de um `dataframe` do tipo `tibble` ocorre a partir da função `tibble`. Note que a criação de um `dataframe` nativo ocorre com a função `base::data.frame`, enquanto a criação do `tibble` parte da função `tibble::tibble` ou `dplyr::tibble`. Para manter o código mais limpo, iremos dar preferência a `dplyr::tibble` e utilizar

o nome *dataframe* para se referir a um *tibble*. Veja o exemplo a seguir, onde criamos uma tabela correspondente a dados financeiros de diferentes ações.

```
library(tibble)

# set tickers
ticker <- c(rep('ABEV3',4),
            rep('BBAS3', 4),
            rep('BBDC3', 4))

# set dates
ref_date <- as.Date(rep(c('2010-01-01', '2010-01-04',
                          '2010-01-05', '2010-01-06'),
                        3) )

# set prices
price <- c(736.67, 764.14, 768.63, 776.47,
           59.4 , 59.8 , 59.2 , 59.28,
           29.81 , 30.82 , 30.38 , 30.20)

# create tibble/dataframe
my_df <- tibble(ticker, ref_date , price)

# print it
print(my_df)
```

```
R> # A tibble: 12 x 3
R>   ticker ref_date price
R>   <chr> <date>   <dbl>
R> 1 ABEV3 2010-01-01 737.
R> 2 ABEV3 2010-01-04 764.
R> 3 ABEV3 2010-01-05 769.
R> 4 ABEV3 2010-01-06 776.
R> 5 BBAS3 2010-01-01  59.4
R> 6 BBAS3 2010-01-04  59.8
R> 7 BBAS3 2010-01-05  59.2
R> 8 BBAS3 2010-01-06  59.3
R> 9 BBDC3 2010-01-01  29.8
R> 10 BBDC3 2010-01-04  30.8
R> 11 BBDC3 2010-01-05  30.4
R> 12 BBDC3 2010-01-06  30.2
```

Observe que utilizamos a função `rep` para replicar e facilitar a criação dos dados

do `dataframe` anterior. Assim, não é necessário repetir os valores múltiplas vezes. Destaca-se que, no uso dos `dataframes`, podemos salvar todos os nossos dados em um único objeto, facilitando o acesso e a organização do código resultante.



O conteúdo de `dataframes` também pode ser visualizado no próprio RStudio. Para isso, basta clicar no nome do objeto na aba *environment*, canto superior direito da tela. Após isso, um visualizador aparecerá na tela principal do programa. Essa operação é nada mais que uma chamada a função `utils::View`. Portanto, poderíamos visualizar o `dataframe` anterior executando o comando `View(my_df)`.

### 6.1.2 Inspeccionando um `dataframe`

Após a criação do `dataframe`, o segundo passo é conhecer o seu conteúdo. Particularmente, é importante tomar conhecimento dos seguintes itens em ordem de importância:

**Número de linhas e colunas** O número de linhas e colunas da tabela resultante indicam se a operação de importação foi executada corretamente. Caso os valores forem diferentes do esperado, deve-se checar o arquivo de importação dos dados e se as opções de importação fazem sentido para o arquivo.

**Nomes das colunas** É importante que a tabela importada tenha nomes que façam sentido e que sejam fáceis de acessar. Portanto, o segundo passo na inspeção de um `dataframe` é analisar os nomes das colunas e seus respectivos conteúdos. Confirme que cada coluna realmente apresenta um nome intuitivo.

**Classes das colunas** Cada coluna de um `dataframe` tem sua própria classe. É de suma importância que as classes dos dados estejam corretamente especificadas. Caso contrário, operações futuras podem resultar em um erro. Por exemplo, caso um vetor de valores numéricos seja importado com a classe de texto (`character`), qualquer operação matemática nesse vetor irá resultar em um erro no R.

**Existência de dados omissos (NA)** Devemos também verificar o número de valores NA (*not available*) nas diferentes colunas. Sempre que você encontrar uma grande proporção de valores NA na tabela importada, você deve descobrir o que está acontecendo e se a informação está sendo importada corretamente. Conforme mencionado no capítulo anterior, os valores NA são contagiosos e transformarão qualquer objeto que interagir com um NA, também se tornará um NA.

Uma das funções mais recomendadas para se familiarizar com um `dataframe` é `dplyr::glimpse`. Essa mostra na tela o nome e a classe das colunas, além do número de linhas/colunas. Abusamos dessa função nos capítulos anteriores. Veja

um exemplo simples a seguir:

```
library(dplyr)

# check content of my_df
glimpse(my_df)
```

```
R> Rows: 12
R> Columns: 3
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
```

Em muitas situações, o uso de `glimpse` é suficiente para entender se o processo de importação de dados ocorreu de forma satisfatória. Porém, uma análise mais profunda é entender qual a variação de cada coluna nos dados importados. Aqui entra o papel da função `base::summary`:

```
# check variation my_df
summary(my_df)
```

```
R>      ticker          ref_date          price
R> Length:12      Min.   :2010-01-01   Min.    : 29.81
R> Class :character 1st Qu.:2010-01-03   1st Qu.: 30.71
R> Mode  :character Median :2010-01-04   Median : 59.34
R>                               Mean  :2010-01-04   Mean   :283.73
R>                               3rd Qu.:2010-01-05   3rd Qu.:743.54
R>                               Max.   :2010-01-06   Max.   :776.47
```

Note que `summary` interpreta cada coluna de forma diferente. Para o primeiro caso, coluna `ticker`, mostra apenas o tamanho do vetor. No caso de datas e valores numéricos, essa apresenta o máximo, mínimo, mediana e quartis. Por exemplo, uma observação extrema (*outlier*) poderia ser facilmente identificada na análise da saída textual de `summary`.



Toda vez que se deparar com um novo `dataframe` no R, pegue o hábito de verificar o seu conteúdo com funções `dplyr::glimpse` e `base::summary`. Assim, poderá perceber problemas de importação e/ou conteúdo dos arquivos lidos. Com experiência irás perceber que muitos erros futuros em código podem ser sanados por uma simples inspeção das tabelas importadas.

### 6.1.3 Operador de *pipeline* (%>%)

Uma característica importante do universo `tidyverse` é o uso extensivo do operador de *pipeline*, primeiro proposto por Bache and Wickham (2022) e definido pelo símbolo `%>%`. Esse comando permite que operações de dados sejam realizadas de forma sequencial e modular, como em uma *tubulação*, facilitando a otimização e legibilidade do código resultante.

Imagine uma situação onde temos três funções para aplicar nos dados salvos em um `dataframe`. Cada função depende da saída de outra função. Isso requer o encadeamento de suas chamadas. Usando o operador de *pipeline*, podemos escrever o procedimento de manipulação `dataframe` com o seguinte código:

```
my_tab <- my_df %>%  
  fct1(arg1) %>%  
  fct2(arg2) %>%  
  fct3(arg3)
```

Usamos símbolo `%>%` no final de cada linha para vincular as operações. As funções `fct*` são operações realizadas em cada etapa. O resultado de cada linha é passado para a próxima função de forma sequencial. Assim, não há necessidade de criar objetos intermediários. Veja a seguir duas formas alternativas de realizar a mesma operação sem o operador de *pipeline*:

```
# version 1  
my_tab <- fct3(fct2(fct1(my_df,  
                        arg1),  
              arg2),  
             arg1)  
  
# version 2  
temp1 <- fct1(my_df, arg1)  
temp2 <- fct2(temp1, arg2)  
my_tab <- fct3(temp1, arg3)
```

Observe como as alternativas formam um código com estrutura estranha e passível a erros. Provavelmente não deves ter notado, mas ambos os códigos possuem erros de digitação. Para o primeiro, o último `arg1` deveria ser `arg3` e, no segundo, a função `fct3` está usando o `dataframe` `temp1` e não `temp2`. Este exemplo deixa claro como o uso de *pipelines* torna o código mais elegante e legível. A partir de agora iremos utilizar o operador `%>%` de forma extensiva.

### 6.1.4 Acessando Colunas

Um objeto do tipo `dataframe` utiliza-se de diversos comandos e símbolos que também são usados em matrizes e listas. Para descobrir os nomes das colunas de um `dataframe`, temos duas funções: `names` ou `colnames`:

```
# check names of df
names(my_df)
```

```
R> [1] "ticker" "ref_date" "price"
```

```
colnames(my_df)
```

```
R> [1] "ticker" "ref_date" "price"
```

Ambas também podem ser usadas para modificar os nomes das colunas:

```
# set temp df
temp_df <- my_df
```

```
# check names
names(temp_df)
```

```
R> [1] "ticker" "ref_date" "price"
```

```
# change names
names(temp_df) <- paste0('Col', 1:ncol(temp_df))
```

```
# check names
names(temp_df)
```

```
R> [1] "Col1" "Col2" "Col3"
```

Destaca-se que a forma de usar `names` é bastante distinta das demais funções do R. Nesse caso, utilizamos a função ao lado esquerdo do símbolo de `assign` (`<-`). Internamente, o que estamos fazendo é definindo um atributo do objeto `temp_df`, o nome de suas colunas.

Para acessar uma determinada coluna, podemos utilizar o nome da mesma:

```
# isolate columns of df
my_ticker <- my_df$ticker
my_prices <- my_df[['price']]
```

```
# print contents
print(my_ticker)
```

```
R> [1] "ABEV3" "ABEV3" "ABEV3" "ABEV3" "BBAS3" "BBAS3" "BBAS3"
```

```
R> [8] "BBAS3" "BBDC3" "BBDC3" "BBDC3" "BBDC3"
```

```
print(my_prices)
```

```
R> [1] 736.67 764.14 768.63 776.47 59.40 59.80 59.20 59.28
R> [9] 29.81 30.82 30.38 30.20
```

Note o uso do duplo colchetes (`[[ ]]`) para selecionar colunas. Vale apontar que, no R, **um objeto da classe `dataframe` é representado internamente como uma lista**, onde cada elemento é uma coluna. Isso é importante saber, pois alguns comandos de listas também funcionam para `dataframes`. Um exemplo é o uso de duplo colchetes (`[[ ]]`) para selecionar colunas por posição:

```
print(my_df[[2]])
```

```
R> [1] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
R> [5] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
R> [9] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

Para acessar linhas e colunas específicas de um `dataframe`, basta utilizar colchetes simples:

```
print(my_df[1:5,2])
```

```
R> # A tibble: 5 x 1
R>   ref_date
R>   <date>
R> 1 2010-01-01
R> 2 2010-01-04
R> 3 2010-01-05
R> 4 2010-01-06
R> 5 2010-01-01
```

```
print(my_df[1:5,c(1,2)])
```

```
R> # A tibble: 5 x 2
R>   ticker ref_date
R>   <chr>  <date>
R> 1 ABEV3  2010-01-01
R> 2 ABEV3  2010-01-04
R> 3 ABEV3  2010-01-05
R> 4 ABEV3  2010-01-06
R> 5 BBAS3  2010-01-01
```

```
print(my_df[1:5, ])
```

```
R> # A tibble: 5 x 3
```

```
R>  ticker ref_date  price
R>  <chr>  <date>    <dbl>
R> 1 ABEV3  2010-01-01 737.
R> 2 ABEV3  2010-01-04 764.
R> 3 ABEV3  2010-01-05 769.
R> 4 ABEV3  2010-01-06 776.
R> 5 BBAS3  2010-01-01  59.4
```

Essa seleção de colunas também pode ser realizada utilizando o nome das mesmas da seguinte forma:

```
print(my_df[1:3, c('ticker', 'price')])
```

```
R> # A tibble: 3 x 2
R>  ticker price
R>  <chr>  <dbl>
R> 1 ABEV3   737.
R> 2 ABEV3   764.
R> 3 ABEV3   769.
```

ou, pelo operador de *pipeline* e a função `dplyr::select`:

```
library(dplyr)

my.temp <- my_df %>%
  select(ticker, price) %>%
  glimpse()
```

```
R> Rows: 12
R> Columns: 2
R> $ ticker <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS3"~
R> $ price <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59.8~
```

### 6.1.5 Modificando um dataframe

Para criar novas colunas em um dataframe, basta utilizar a função `mutate`. Aqui iremos abusar do operador de *pipeline* (`%>%`) para sequenciar as operações:

```
library(dplyr)

# add columns with mutate
my_df <- my_df %>%
  mutate(ret = price/lag(price) -1,
         my_seq1 = 1:nrow(my_df),
         my_seq2 = my_seq1 +9) %>%
  glimpse()
```

```
R> Rows: 12
R> Columns: 6
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
R> $ my_seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2~
```

Note que precisamos indicar o `dataframe` de origem dos dados, nesse caso o objeto `my_df`, e as colunas são definidas como argumentos em `dplyr::mutate`. Observe também que usamos a coluna `price` na construção de `ret`, o retorno aritmético dos preços. Um caso especial é a construção de `my_seq2` com base em `my_seq1`, isto é, antes mesmo dela ser explicitamente calculada já é possível utilizar a nova coluna para criar outra. Vale salientar que a nova coluna deve ter exatamente o mesmo número de elementos que as demais. Caso contrário, o R retorna uma mensagem de erro.

A maneira mais tradicional, e comumente encontrada em código, para criar novas colunas é utilizar o símbolo `$`:

```
# add new column with base R
my_df$my_seq3 <- 1:nrow(my_df)

# check it
glimpse(my_df)
```

```
R> Rows: 12
R> Columns: 7
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
R> $ my_seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2~
R> $ my_seq3  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Portanto, o operador `$` vale tanto para acessar quanto para criar novas colunas.

Para remover colunas de um `dataframe`, basta usar `dplyr::select` com operador negativo para o nome das colunas indesejadas:

```
# removing columns
my_df_temp <- my_df %>%
```

```
select(-my_seq1, -my_seq2, -my_seq3) %>%
  glimpse()
```

```
R> Rows: 12
R> Columns: 4
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS3~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
```

No uso de funções nativas do R, a maneira tradicional de remover colunas é alocar o valor nulo (NULL):

```
# set temp df
temp_df <- my_df

# remove cols
temp_df$price <- NULL
temp_df$ref_date <- NULL

# check it
glimpse(temp_df)
```

```
R> Rows: 12
R> Columns: 5
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS3~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966,~
R> $ my_seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20~
R> $ my_seq3  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

### 6.1.6 Filtrando um dataframe

Uma operação bastante comum no R é filtrar linhas de uma tabela de acordo com uma ou mais condições. Por exemplo, caso quiséssemos apenas os dados da ação ABEV3, poderíamos utilizar a função `dplyr::filter` para filtrar a tabela:

```
library(dplyr)

# filter df for single stock
my_df_temp <- my_df %>%
  filter(ticker == 'ABEV3') %>%
  glimpse()
```

```
R> Rows: 4
R> Columns: 7
R> $ ticker <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3"
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010-~
R> $ price <dbl> 736.67, 764.14, 768.63, 776.47
R> $ ret <dbl> NA, 0.037289424, 0.005875887, 0.010199966
R> $ my_seq1 <int> 1, 2, 3, 4
R> $ my_seq2 <dbl> 10, 11, 12, 13
R> $ my_seq3 <int> 1, 2, 3, 4
```

A função também aceita mais de uma condição. Veja a seguir onde filtramos os dados para 'ABEV3' em datas após ou igual a '2010-01-05':

```
library(dplyr)
# filter df for single stock and date
my_df_temp <- my_df %>%
  filter(ticker == 'ABEV3',
         ref_date >= as.Date('2010-01-05')) %>%
  glimpse()
```

```
R> Rows: 2
R> Columns: 7
R> $ ticker <chr> "ABEV3", "ABEV3"
R> $ ref_date <date> 2010-01-05, 2010-01-06
R> $ price <dbl> 768.63, 776.47
R> $ ret <dbl> 0.005875887, 0.010199966
R> $ my_seq1 <int> 3, 4
R> $ my_seq2 <dbl> 12, 13
R> $ my_seq3 <int> 3, 4
```

Aqui utilizamos o símbolo == para testar uma igualdade. Iremos estudar mais profundamente a classe de testes lógicos no capítulo 7.

### 6.1.7 Ordenando um dataframe

Após a criação ou importação de um **dataframe**, pode-se ordenar seus componentes de acordo com os valores de alguma coluna. Um caso bastante comum em que é necessário realizar uma ordenação explícita é quando importamos dados financeiros em que as datas não estão em ordem crescente. Na grande maioria das situações, dados temporais devem estar ordenados de acordo com a antiguidade, isto é, dados mais recentes são alocados na última linha da tabela. Essa operação é realizada através do uso da função `base::order` ou `dplyr::arrange`.

Como exemplo, considere a criação de um **dataframe** com os valores a seguir:

```
library(tidyverse)

# set df
my_df <- tibble(col1 = c(4,1,2),
                col2 = c(1,1,3),
                col3 = c('a','b','c'))

# print it
print(my_df)
```

```
R> # A tibble: 3 x 3
R>   col1 col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     1     1 b
R> 3     2     3 c
```

A função `order` retorna os índices relativos à ordenação dos valores dados como entrada. Para o caso da primeira coluna de `my_df`, os índices dos elementos formadores do novo vetor, com seus valores ordenados em forma crescente, são:

```
idx <- order(my_df$col1)
print(idx)
```

```
R> [1] 2 3 1
```

Portanto, ao utilizar a saída da função `order` como indexador do `dataframe`, acaba-se ordenando o mesmo de acordo com os valores da coluna `col1`. Veja a seguir:

```
my_df_2 <- my_df[order(my_df$col1), ]
print(my_df_2)
```

```
R> # A tibble: 3 x 3
R>   col1 col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     2     3 c
R> 3     4     1 a
```

Essa operação de ordenamento também pode ser realizada levando em conta mais de uma coluna. Veja o exemplo a seguir, onde se ordena o `dataframe` pelas colunas `col2` e `col1`.

```
idx <- order(my_df$col2, my_df$col1)
my_df_3 <- my_df[idx, ]
```

```
print(my_df_3)
```

```
R> # A tibble: 3 x 3
R>   col1 col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     4     1 a
R> 3     2     3 c
```

No tidyverse, a forma de ordenar **dataframes** é pelo uso da função `arrange`. No caso de ordenamento decrescente, encapsulamos o nome das colunas com `desc`:

```
# sort ascending, by col1 and col2
my_df <- my_df %>%
  arrange(col1, col2) %>%
  print()
```

```
R> # A tibble: 3 x 3
R>   col1 col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     2     3 c
R> 3     4     1 a
```

```
# sort descending, col1 and col2
my_df <- my_df %>%
  arrange(desc(col1), desc(col2)) %>%
  print()
```

```
R> # A tibble: 3 x 3
R>   col1 col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     2     3 c
R> 3     1     1 b
```

O resultado prático no uso de `arrange` é o mesmo de `order`. Um dos seus benefícios é a possibilidade de encadeamento de operações através do uso do *pipeline*.

### 6.1.8 Combinando e Agregando dataframes

Em muitas situações de análise de dados será necessário juntar **dataframes** distintos em um único objeto. Tabelas diferentes são importadas no R e, antes de analisar os dados, precisamos combinar as informações em um único objeto. Nos casos mais simples, onde as tabelas a serem agregadas possuem o mesmo formato, nós as

juntamos de acordo com as linhas, verticalmente, ou colunas, horizontalmente. Para esse fim, temos as funções `dplyr::bind_rows` e `dplyr::bind_cols` no tidyverse e `base::rbind` e `base::cbind` nas funções nativas do R. Observe o exemplo a seguir.

```
library(dplyr)

# set dfs
my_df_1 <- tibble(col1 = 1:5,
                  col2 = rep('a', 5))

my_df_2 <- tibble(col1 = 6:10,
                  col2 = rep('b', 5),
                  col3 = rep('c', 5))

# bind by row
my_df <- bind_rows(my_df_1, my_df_2) %>%
  glimpse()
```

```
R> Rows: 10
R> Columns: 3
R> $ col1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
R> $ col2 <chr> "a", "a", "a", "a", "a", "b", "b", "b", "b", ~
R> $ col3 <chr> NA, NA, NA, NA, NA, "c", "c", "c", "c", "c"
```

Note que, no exemplo anterior, os nomes das colunas são os mesmos. De fato, a função `dplyr::bind_rows` procura os nomes iguais em ambos os objetos para fazer a junção dos dataframes corretamente. As colunas que não ocorrem em ambos objetos, tal como `col3` no exemplo, saem como `NA` no objeto final. Já para o caso de `bind_cols`, os nomes das colunas devem ser diferentes, porém o número de linhas deve ser o mesmo.

```
# set dfs
my_df_1 <- tibble(col1 = 1:5, col2 = rep('a', 5))
my_df_2 <- tibble(col3 = 6:10, col4 = rep('b', 5))

# bind by column
my_df <- bind_cols(my_df_1, my_df_2) %>%
  glimpse()
```

```
R> Rows: 5
R> Columns: 4
R> $ col1 <int> 1, 2, 3, 4, 5
R> $ col2 <chr> "a", "a", "a", "a", "a"
```

```
R> $ col3 <int> 6, 7, 8, 9, 10
R> $ col4 <chr> "b", "b", "b", "b", "b"
```

Para casos mais complexos, onde a junção deve ser realizada de acordo com algum índice tal como uma data, é possível juntar `dataframes` diferentes com o uso das funções da família `dplyr::join*` tal como `dplyr::inner_join` `dplyr::left_join`, `dplyr::full_join`, entre outras. A descrição de todas elas não cabe aqui. Iremos descrever apenas o caso mais provável, `inner_join`. Essa combina os dados, mantendo apenas os casos onde existe o índice em ambos.

```
# set df
my_df_1 <- tibble(date = as.Date('2016-01-01')+0:10,
                  x = 1:11)

my_df_2 <- tibble(date = as.Date('2016-01-05')+0:10,
                  y = seq(20,30, length.out = 11))
```

Note que os `dataframes` criados possuem uma coluna em comum, `date`. A partir desta coluna que agregamos as tabelas com `inner_join`:

```
# aggregate tables
my_df <- inner_join(my_df_1, my_df_2)
```

```
R> Joining, by = "date"
```

```
glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-01--
R> $ x    <int> 5, 6, 7, 8, 9, 10, 11
R> $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

O R automaticamente verifica a existência de colunas com mesmo nome nos `dataframes` e realiza a junção por essas. Caso quiséssemos juntar `dataframes` onde os nomes das colunas para utilizar o índice não são iguais, temos duas soluções: modificar os nomes das colunas ou então utilizar argumento `by` em `dplyr::inner_join`. Veja a seguir:

```
# set df
my_df_3 <- tibble(ref_date = as.Date('2016-01-01')+0:10,
                  x = 1:11)

my_df_4 <- tibble(my_date = as.Date('2016-01-05')+0:10,
                  y = seq(20,30, length.out = 11))
```

```
# join by my_df_3$ref_date and my_df_4$my_date
my_df <- inner_join(my_df_3, my_df_4,
                   by = c('ref_date' = 'my_date'))

glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ ref_date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-
R> $ x      <int> 5, 6, 7, 8, 9, 10, 11
R> $ y      <dbl> 20, 21, 22, 23, 24, 25, 26
```

Para o caso de uso da função nativa de agregação de dataframes, `base::merge`, temos que indicar explicitamente o nome da coluna com argumento `by`:

```
# aggregation with base R
my_df <- merge(my_df_1, my_df_2, by = 'date')

glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-01-~
R> $ x    <int> 5, 6, 7, 8, 9, 10, 11
R> $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

Note que, nesse caso, o `dataframe` resultante manteve apenas as informações compartilhadas entre ambos os objetos, isto é, aquelas linhas onde as datas em `date` eram iguais. Esse é o mesmo resultado quando no uso do `dplyr::inner_join`.

As demais funções de agregação de tabelas – `left_join`, `right_join`, `outer_join` e `full_join` – funcionam de forma muito semelhante a `inner_join`, exceto na escolha da saída. Por exemplo, `full_join` retorna todos os casos/linhas entre tabela 1 e 2, incluindo aqueles onde não tem o índice compartilhado. Para estes casos, a coluna do índice sairá como NA. Veja o exemplo a seguir:

```
# set df
my_df_5 <- tibble(ref_date = as.Date('2016-01-01')+0:10,
                 x = 1:11)

my_df_6 <- tibble(ref_date = as.Date('2016-01-05')+0:10,
                 y = seq(20,30, length.out = 11))

# combine with full_join
my_df <- full_join(my_df_5, my_df_6)
```

```
R> Joining, by = "ref_date"
```

```
# print it
print(my_df)
```

```
R> # A tibble: 15 x 3
R>   ref_date      x      y
R>   <date>    <int> <dbl>
R> 1 2016-01-01     1    NA
R> 2 2016-01-02     2    NA
R> 3 2016-01-03     3    NA
R> 4 2016-01-04     4    NA
R> 5 2016-01-05     5    20
R> 6 2016-01-06     6    21
R> 7 2016-01-07     7    22
R> 8 2016-01-08     8    23
R> 9 2016-01-09     9    24
R> 10 2016-01-10    10    25
R> 11 2016-01-11    11    26
R> 12 2016-01-12    NA    27
R> 13 2016-01-13    NA    28
R> 14 2016-01-14    NA    29
R> 15 2016-01-15    NA    30
```

### 6.1.9 Extensões ao dataframe

Como já foi relatado nos capítulos anteriores, um dos grandes benefícios no uso do R é a existência de pacotes para lidar com os problemas específicos dos usuários. Enquanto um objeto tabular do tipo `tibble` é suficiente para a maioria dos casos, existem benefícios no uso de uma classe alternativa. Ao longo do tempo, diversas soluções foram disponibilizadas por desenvolvedores.

Por exemplo, é muito comum trabalharmos com dados exclusivamente numéricos que são indexados ao tempo. Isto é, situações onde cada informação pertence a um índice temporal - um objeto da classe `data/tempo`. As linhas dessa tabela representam um ponto no tempo, enquanto as colunas indicam variáveis numéricas de interesse. Nesse caso, faria sentido representarmos os nossos dados como objetos do tipo `xts` (Ryan and Ulrich, 2022b). O grande benefício dessa opção é que a agregação e a manipulação de variáveis em função do tempo é muito fácil. Por exemplo, podemos transformar dados de frequência diária para a frequência semanal com apenas uma linha de comando. Além disso, diversas outras funções reconhecem automaticamente que os dados são indexados ao tempo. Um exemplo é a criação de uma figura com esses dados. Nesse caso, o eixo horizontal da figura é automaticamente organizado com as datas.

Veja um caso a seguir, onde carregamos os dados anteriores como um objeto `xts`:

```
library(xts)

# set data
ticker <- c('ABEV3', 'BBAS3', 'BBDC3')

date <- as.Date(c('2010-01-01', '2010-01-04',
                 '2010-01-05', '2010-01-06'))

price_ABEV3 <- c(736.67, 764.14, 768.63, 776.47)
price_BBAS3 <- c(59.4, 59.8, 59.2, 59.28)
price_BBDC3 <- c(29.81, 30.82, 30.38, 30.20)

# build matrix
my_mat <- matrix(c(price_BBDC3, price_BBAS3, price_ABEV3),
                nrow = length(date) )

# set xts object
my_xts <- xts(my_mat,
              order.by = date)

# set correct colnames
colnames(my_xts) <- ticker

# check it!
print(my_xts)
```

```
R>           ABEV3 BBAS3  BBDC3
R> 2010-01-01 29.81 59.40 736.67
R> 2010-01-04 30.82 59.80 764.14
R> 2010-01-05 30.38 59.20 768.63
R> 2010-01-06 30.20 59.28 776.47
```

O código anterior pode dar a impressão de que o objeto `my_xts` é semelhante a um `dataframe`, porém, não se engane. Por estar indexado a um vetor de tempo, objeto `xts` pode ser utilizado para uma série de procedimentos temporais, tal como uma agregação por período temporal. Veja o exemplo a seguir, onde agregamos duas variáveis de tempo através do cálculo de uma média a cada semana.

```
N <- 500

my_mat <- matrix(c(seq(1, N), seq(N, 1)), nrow=N)
```

```
my_xts <- xts(my_mat, order.by = as.Date('2016-01-01')+1:N)

my_xts.weekly.mean <- apply.weekly(my_xts, mean)

print(head(my_xts.weekly.mean))
```

```
R>           X.1  X.2
R> 2016-01-03  1.5 499.5
R> 2016-01-10  6.0 495.0
R> 2016-01-17 13.0 488.0
R> 2016-01-24 20.0 481.0
R> 2016-01-31 27.0 474.0
R> 2016-02-07 34.0 467.0
```

Em Finanças e Economia, as agregações com objetos `xts` são extremamente úteis quando se trabalha com dados em frequências de tempo diferentes. Por exemplo, é muito comum que se agregue dados de transação no mercado financeiro em alta frequência para intervalos maiores. Assim, dados que ocorrem a cada segundo são agregados para serem representados de 15 em 15 minutos. Esse tipo de procedimento é facilmente realizado no R através da correta representação dos dados como objetos `xts`. Existem diversas outras funcionalidades desse pacote. Encorajo os usuários a ler o manual e aprender o que pode ser feito.

Indo além, existem diversos outros tipos de `dataframes` customizados. Por exemplo, o `dataframe` proposto pelo pacote `data.table` (Dowle and Srinivasan, 2022) prioriza o tempo de operação nos dados e o uso de uma notação compacta para acesso e processamento. O `tibbletime` (Vaughan and Dancho, 2022) é uma versão orientada pelo tempo para `tibbles`. Caso o usuário esteja necessitando realizar operações de agregação de tempo, o uso deste pacote é fortemente recomendado.

### 6.1.10 Outras Funções Úteis

**head** - Retorna os primeiros `n` elementos de um `dataframe`.

```
my_df <- tibble(col1 = 1:5000, col2 = rep('a', 5000))
head(my_df, 5)
```

```
R> # A tibble: 5 x 2
R>   col1 col2
R>   <int> <chr>
R> 1     1  a
R> 2     2  a
R> 3     3  a
R> 4     4  a
```

```
R> 5      5 a
```

**tail** - Retorna os últimos *n* elementos de um `dataframe`.

```
tail(my_df, 5)
```

```
R> # A tibble: 5 x 2
R>   col1 col2
R>   <int> <chr>
R> 1  4996 a
R> 2  4997 a
R> 3  4998 a
R> 4  4999 a
R> 5  5000 a
```

**complete.cases** - Retorna um vetor lógico que testa se as linhas contêm apenas valores existentes e nenhum `NA`.

```
my_x <- c(1:5, NA, 10)
my_y <- c(5:10, NA)
my_df <- tibble(my_x, my_y)
```

```
print(my_df)
```

```
R> # A tibble: 7 x 2
R>   my_x my_y
R>   <dbl> <int>
R> 1     1     5
R> 2     2     6
R> 3     3     7
R> 4     4     8
R> 5     5     9
R> 6    NA    10
R> 7    10    NA
```

```
print(complete.cases(my_df))
```

```
R> [1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE
```

```
print(which(!complete.cases(my_df)))
```

```
R> [1] 6 7
```

**na.omit** - Retorna um `dataframe` sem as linhas onde valores `NA` são encontrados.

```
print(na.omit(my_df))
```

```
R> # A tibble: 5 x 2
R>   my_x my_y
R>   <dbl> <int>
R> 1     1     5
R> 2     2     6
R> 3     3     7
R> 4     4     8
R> 5     5     9
```

**unique** - Retorna um **dataframe** onde todas as linhas duplicadas são eliminadas e somente os casos únicos são mantidos.

```
my_df <- tibble(col1 = c(1,1,2,3,3,4,5),
                col2 = c('A','A','A','C','C','B','D'))

print(my_df)
```

```
R> # A tibble: 7 x 2
R>   col1 col2
R>   <dbl> <chr>
R> 1     1 A
R> 2     1 A
R> 3     2 A
R> 4     3 C
R> 5     3 C
R> 6     4 B
R> 7     5 D
```

```
print(unique(my_df))
```

```
R> # A tibble: 5 x 2
R>   col1 col2
R>   <dbl> <chr>
R> 1     1 A
R> 2     2 A
R> 3     3 C
R> 4     4 B
R> 5     5 D
```

## 6.2 Listas

Uma lista (**list**) é uma classe de objeto extremamente flexível e já tivemos contato com ela nos capítulos anteriores. Ao contrário de vetores atômicos, a lista não apresenta restrição alguma em relação aos tipos de elementos nela contidos. Po-

demos agrupar valores numéricos com caracteres, fatores com datas e até mesmo listas dentro de listas. Quando agrupamos vetores, também não é necessário que os mesmos tenham um número igual de elementos. Além disso, podemos dar um nome a cada elemento. Essas propriedades fazem da lista o objeto mais flexível para o armazenamento e estruturação de dados no R. Não é acidental o fato de que listas são muito utilizadas como retorno de funções.

### 6.2.1 Criando Listas

Uma lista pode ser criada através do comando `base::list`, seguido por seus elementos separados por vírgula:

```
library(dplyr)

# create list
my_l <- list(c(1, 2, 3),
             c('a', 'b'),
             factor('A', 'B', 'C'),
             tibble(col1 = 1:5))

# use base::print
print(my_l)
```

```
R> [[1]]
R> [1] 1 2 3
R>
R> [[2]]
R> [1] "a" "b"
R>
R> [[3]]
R> [1] <NA>
R> Levels: C
R>
R> [[4]]
R> # A tibble: 5 x 1
R>   col1
R>   <int>
R> 1     1
R> 2     2
R> 3     3
R> 4     4
R> 5     5
```

```
# use dplyr::glimpse
glimpse(my_l)
```

```
R> List of 4
R> $ : num [1:3] 1 2 3
R> $ : chr [1:2] "a" "b"
R> $ : Factor w/ 1 level "C": NA
R> $ : tibble [5 x 1] (S3: tbl_df/tbl/data.frame)
R> ..$ col1: int [1:5] 1 2 3 4 5
```

Note que juntamos no mesmo objeto um vetor atômico numérico, outro de texto, um fator e um `tibble`. A apresentação de listas com o comando `print` é diferente dos casos anteriores. Os elementos são separados verticalmente e os seus índices aparecem com duplo colchete (`[[ ]]`). Conforme será explicado logo a seguir, é dessa forma que os elementos de uma lista são armazenados e acessados.

Assim como para os demais tipos de objeto, os elementos de uma lista também podem ter nomes, o que facilita o entendimento e a interpretação das informações do problema em análise. Por exemplo, considere o caso de uma base de dados com informações sobre determinada ação negociada na bolsa. Nesse caso, podemos definir uma lista como:

```
# set named list
my_named_l <- list(ticker = 'TICK4',
                  market = 'Bovespa',
                  df_prices = tibble(P = c(1,1.5,2,2.3),
                                     ref_date = Sys.Date()+0:3))

# check content
glimpse(my_named_l)
```

```
R> List of 3
R> $ ticker : chr "TICK4"
R> $ market : chr "Bovespa"
R> $ df_prices: tibble [4 x 2] (S3: tbl_df/tbl/data.frame)
R> ..$ P : num [1:4] 1 1.5 2 2.3
R> ..$ ref_date: Date[1:4], format: "2022-11-23" ...
```



Toda vez que for trabalhar com listas, facilite a sua vida ao nomear todos os elementos de forma intuitiva. Isso facilita o acesso e evita possíveis erros no código.

### 6.2.2 Acessando os Elementos de uma Lista

Os elementos de uma lista podem ser acessados através do uso de duplo colchete (`[[ ]]`), tal como em:

```
# accessing elements from list
print(my_named_1[[2]])
```

```
R> [1] "Bovespa"
```

```
print(my_named_1[[3]])
```

```
R> # A tibble: 4 x 2
R>       P ref_date
R>   <dbl> <date>
R> 1     1 2022-11-23
R> 2   1.5 2022-11-24
R> 3     2 2022-11-25
R> 4   2.3 2022-11-26
```

Também é possível acessar os elementos com um colchete simples (`[ ]`), porém, tome cuidado com essa operação, pois o resultado não vai ser o objeto em si, mas uma outra lista. Esse é um equívoco muito fácil de passar despercebido, resultando em erros no código. Veja a seguir:

```
# set list
my_l <- list('a',
             c(1,2,3),
             factor('a','b'))
```

```
# check classes
class(my_l[[2]])
```

```
R> [1] "numeric"
```

```
class(my_l[2])
```

```
R> [1] "list"
```

Caso tentarmos somar um elemento a `my_l[2]`, teremos uma mensagem de erro:

```
my_l[2] + 1
```

```
R> Error in my_l[2] + 1: non-numeric argument to binary operator
```

Esse erro ocorre devido ao fato de que uma lista não tem operador de soma. Para corrigir, basta utilizar o duplo colchete, tal como em `my_l[[2]]+1`. O acesso a elementos de uma lista com colchete simples somente é útil quando estamos procurando

uma sublista dentro de uma lista maior. No exemplo anterior, caso quiséssemos obter o primeiro e o segundo elemento da lista `my_1`, usaríamos:

```
# set new list
my_new_1 <- my_1[c(1,2)]

# check contents
print(my_new_1)
```

```
R> [[1]]
R> [1] "a"
R>
R> [[2]]
R> [1] 1 2 3
```

```
class(my_new_1)
```

```
R> [1] "list"
```

No caso de listas com elementos nomeados, os mesmos podem ser acessados por seu nome através do uso do símbolo `$` tal como em `my_named_1$df_prices` ou `[[ 'nome' ]]`, tal como em `my_named_1[[ 'df_prices' ]]`. Em geral, essa é uma forma mais eficiente e recomendada de interagir com os elementos de uma lista. Como regra geral no uso do R, sempre dê preferência ao acesso de elementos através de seus nomes, seja em listas, vetores ou **dataframes**. Isso evita erros, pois, ao modificar os dados e adicionar algum outro objeto na lista, é possível que o ordenamento interno mude e, portanto, a posição de determinado objeto pode acabar sendo modificada.



Saiba que a ferramenta de *autocomplete* do RStudio também funciona para listas. Para usar, digite o nome da lista seguido de `$` e aperte *tab*. Uma caixa de diálogo com todos os elementos disponíveis na lista irá aparecer. A partir disso, basta selecionar apertando *enter*.

Veja os exemplos a seguir, onde são apresentadas as diferentes formas de se acessar uma lista.

```
# different ways to access a list
my_named_1$ticker
my_named_1$price
my_named_1[['ticker']]
my_named_1[['price']]
```

Vale salientar que também é possível acessar diretamente os elementos de um vetor

que esteja dentro de uma lista através de colchetes encadeados. Veja a seguir:

```
# accessing elements of a vector in a list
my_l <- list(c(1,2,3),
            c('a', 'b'))

print(my_l[[1]][2])
```

```
R> [1] 2
```

```
print(my_l[[2]][1])
```

```
R> [1] "a"
```

Tal operação é bastante útil quando interessa apenas um elemento dentro de um objeto maior criado por alguma função.

### 6.2.3 Adicionando e Removendo Elementos de uma Lista

A remoção, adição e substituição de elementos de uma lista também são procedimentos fáceis. Para adicionar ou substituir, basta definir um novo objeto na posição desejada da lista:

```
# set list
my_l <- list('a', 1, 3)
glimpse(my_l)
```

```
R> List of 3
R> $ : chr "a"
R> $ : num 1
R> $ : num 3
```

```
# add new elements to list
my_l[[4]] <- c(1:5)
my_l[[2]] <- c('b')

# print result
glimpse(my_l)
```

```
R> List of 4
R> $ : chr "a"
R> $ : chr "b"
R> $ : num 3
R> $ : int [1:5] 1 2 3 4 5
```

A operação também é possível com o uso de nomes e operador \$:

```
# set list
my_l <- list(elem1 = 'a', name1=5)

# set new element
my_l$name2 <- 10
glimpse(my_l)
```

```
R> List of 3
R> $ elem1: chr "a"
R> $ name1: num 5
R> $ name2: num 10
```

Para remover elementos de uma lista, basta definir o elemento para o símbolo reservado NULL (nulo):

```
# set list
my_l <- list(text = 'b', num1 = 2, num2 = 4)
glimpse(my_l)
```

```
R> List of 3
R> $ text: chr "b"
R> $ num1: num 2
R> $ num2: num 4
```

```
# remove elements
my_l[[3]] <- NULL
glimpse(my_l)
```

```
R> List of 2
R> $ text: chr "b"
R> $ num1: num 2
```

```
my_l$num1 <- NULL
glimpse(my_l)
```

```
R> List of 1
R> $ text: chr "b"
```

Outra maneira de retirar elementos de uma lista é utilizando um índice negativo para os elementos indesejados. Observe a seguir, onde eliminamos o segundo elemento de uma lista:

```
# set list
my_l <- list(a = 1, b = 'texto')

# remove second element
```

```
glimpse(my_1[[-2]])
```

```
R> num 1
```

Assim como no caso de vetores atômicos, essa remoção também pode ser realizada por condições lógicas. Veja a seguir:

```
# set list
my_1 <- list(1, 2, 3, 4)

# remove elements by condition
my_1[my_1 > 2] <- NULL
glimpse(my_1)
```

```
R> List of 2
```

```
R> $ : num 1
```

```
R> $ : num 2
```

Porém, note que esse atalho só funciona porque todos os elementos de `my_1` são numéricos.

#### 6.2.4 Processando os Elementos de uma Lista

Um ponto importante a ser destacado a respeito de listas é que os seus elementos podem ser processados e manipulados individualmente através de funções específicas. Este é um tópico particular de programação com o R, mas que vale a apresentação aqui.

Por exemplo, imagine uma lista com vetores numéricos de diferentes tamanhos, tal como a seguir:

```
# set list
my_1_num <- list(c(1, 2, 3),
                 seq(1:50),
                 seq(-5, 5, by = 0.5))
```

Caso quiséssemos calcular a média de cada elemento de `my_1_num` e apresentar o resultado na tela como um vetor, poderíamos fazer isso através de um procedimento simples, processando cada elemento individualmente:

```
# calculate mean of vectors
mean_1 <- mean(my_1_num[[1]])
mean_2 <- mean(my_1_num[[2]])
mean_3 <- mean(my_1_num[[3]])
```

```
# print it
print(c(mean_1, mean_2, mean_3))
```

```
R> [1] 2.0 25.5 0.0
```

O código anterior funciona, porém não é recomendado devido sua falta de escalabilidade. Isto é, caso aumentássemos o volume de dados ou objetos, o código não funcionaria corretamente. Se, por exemplo, tivéssemos um quarto elemento em `my_l_num` e quiséssemos manter essa estrutura do código, teríamos que adicionar uma nova linha `mean_4 <- mean(my_l_num[[4]])` e modificar o comando de saída na tela para `print(c(mean_1, mean_2, mean_3, mean_4))`.

Uma maneira mais fácil, elegante e inteligente seria utilizar a função `sapply`. Nela, basta indicar o nome do objeto de tipo lista e a função que queremos utilizar para processar cada elemento. Internamente, os cálculos são realizados automaticamente. Veja a seguir:

```
# using sapply
my_mean <- sapply(my_l_num, mean)

# print result
print(my_mean)
```

```
R> [1] 2.0 25.5 0.0
```

O uso da função `sapply` é preferível por ser mais compacto e eficiente do que a alternativa – a criação de `mean_1`, `mean_2` e `mean_3`. Note que o primeiro código, com médias individuais, só funciona para uma lista com três elementos. A função `sapply`, ao contrário, funcionaria da mesma forma em listas de qualquer tamanho. Caso tivéssemos mais elementos, nenhuma modificação seria necessária no código anterior, o que o torna extensível a chegada de novos dados.

Essa visão e implementação de código voltado a procedimentos genéricos é um dos lemas para tornar o uso do R mais eficiente. **A regra é simples: sempre escreva códigos que sejam adaptáveis a chegada de novos dados.** Em inglês, isso é chamado de regra *DRY* (*don't repeat yourself*). Caso você esteja repetindo códigos e abusando do *control + c/control + v*, como no exemplo anterior, certamente existe uma solução mais elegante e flexível que poderia ser utilizada. No R, existem diversas outras funções da família `apply` para esse objetivo. Essas funções serão explicadas com maiores detalhes no capítulo 8.

### 6.2.5 Outras Funções Úteis

**unlist** - Retorna os elementos de uma lista em um único vetor atômico.

```
my_named_1 <- list(ticker = 'XXXX4',
                  price = c(1,1.5,2,3),
                  market = 'Bovespa')
my_unlisted <- unlist(my_named_1)
print(my_unlisted)
```

```
R>   ticker   price1   price2   price3   price4   market
R>  "XXXX4"    "1"     "1.5"    "2"     "3"  "Bovespa"
```

```
class(my_unlisted)
```

```
R> [1] "character"
```

**as.list** - Converte um objeto para uma lista, tornando cada elemento um elemento da lista.

```
my_x <- 10:13
my_x_as_list <- as.list(my_x)
print(my_x_as_list)
```

```
R> [[1]]
R> [1] 10
R>
R> [[2]]
R> [1] 11
R>
R> [[3]]
R> [1] 12
R>
R> [[4]]
R> [1] 13
```

**names** - Retorna ou define os nomes dos elementos de uma lista. Assim como para o caso de nomear elementos de um vetor atômico, usa-se a função **names** alocada ao lado esquerdo do símbolo **<-**.

```
my_1 <- list(value1 = 1, value2 = 2, value3 = 3)
print(names(my_1))
```

```
R> [1] "value1" "value2" "value3"
```

```
my_1 <- list(1,2,3)
names(my_1) <- c('num1', 'num2', 'num3')
print(my_1)
```

```
R> $num1
```

```
R> [1] 1
R>
R> $num2
R> [1] 2
R>
R> $num3
R> [1] 3
```

### 6.3 Matrizes

Como você deve lembrar das aulas de matemática, uma matriz é uma representação bidimensional de diversos valores, arranjados em linhas e colunas. O uso de matrizes é uma poderosa maneira de representar dados numéricos em duas dimensões e, em certos casos, funções matriciais podem simplificar operações matemáticas complexas.

No R, matrizes são objetos com duas dimensões, onde todos os elementos devem ser da mesma classe. Além disso, as linhas e colunas também podem ter nomes. Assim como para listas e `dataframes`, isso facilita a interpretação e contextualização dos dados.

Um claro exemplo do uso de matrizes em Finanças seria a representação dos preços de diferentes ações ao longo do tempo. Nesse caso, teríamos as linhas representando as diferentes datas e as colunas representando cada ativo, tal como a seguir:

Data	ABEV3	BBAS3	BBDC3
2010-01-01	736.67	59.4	29.81
2010-01-04	764.14	59.8	30.82
2010-01-05	768.63	59.2	30.38
2010-01-06	776.47	59.28	30.20

A matriz anterior poderia ser criada da seguinte forma no R:

```
# create matrix
data <- c(736.67, 764.14, 768.63, 776.47,
         59.4, 59.8, 59.2, 59.28,
         29.81, 30.82, 30.38, 30.20)

my_mat <- matrix(data, nrow = 4, ncol = 3)

# set names of cols and rows
colnames(my_mat) <- c('ABEV3', 'BBAS3', 'BBDC3')
```

```
rownames(my_mat) <- c('2010-01-01', '2010-01-04',
                     '2010-01-05', '2010-01-06')

print(my_mat)
```

```
R>           ABEV3 BBAS3 BBDC3
R> 2010-01-01 736.67 59.40 29.81
R> 2010-01-04 764.14 59.80 30.82
R> 2010-01-05 768.63 59.20 30.38
R> 2010-01-06 776.47 59.28 30.20
```

Observe que, na construção da matriz, definimos o número de linhas e colunas explicitamente com os argumentos `nrow = 4` e `ncol = 3`. Já os nomes das linhas e colunas são definidos pelos comandos `colnames` e `rownames`. Destaca-se, novamente, que a forma de utilizá-los é bastante distinta das demais funções do R. Nesse caso, utilizamos a função ao lado esquerdo do símbolo de `assign (<-)`. Poderíamos, porém, recuperar os nomes das linhas e colunas com as mesmas funções. Veja a seguir:

```
colnames(my_mat)
```

```
R> [1] "ABEV3" "BBAS3" "BBDC3"
```

```
rownames(my_mat)
```

```
R> [1] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

No momento em que temos essa nossa matriz criada, podemos utilizar todas as suas propriedades numéricas. Um exemplo simples é o cálculo do valor de um portfólio de investimento ao longo do tempo. Caso um investidor tenha 200 ações da ABEV3, 300 da BBAS3 e 100 da BBDC3, o valor do seu portfólio ao longo do tempo poderá ser calculado assim:

$$V_t = \sum_{i=1}^3 N_i P_{i,t}$$

Onde  $N_i$  é o número de ações compradas para ativo  $i$  e  $P_{i,t}$  é o preço da ação  $i$  na data  $t$ . Essa é uma operação bastante simples de ser efetuada com uma multiplicação matricial. Traduzindo o procedimento para o R, temos:

```
my.w <- as.matrix(c(200, 300, 100), nrow = 3)
my_port <- my_mat %*% my.w
print(my_port)
```

```
R>           [,1]
R> 2010-01-01 168135
```

```
R> 2010-01-04 173850
R> 2010-01-05 174524
R> 2010-01-06 176098
```

Nesse último exemplo, utilizamos o símbolo `%*%`, o qual permite a multiplicação matricial entre os objetos. O objeto `my_port` indica o valor desse portfólio ao longo das datas, resultando em um leve lucro para o investidor.

Um ponto importante a ressaltar é que uma matriz no R não precisa necessariamente ser composta por valores. É possível, também, criar matrizes de caracteres (texto). Veja o exemplo a seguir:

```
my_mat_char <- matrix(rep(c('a','b','c'), 3) ,
                      nrow = 3,
                      ncol = 3)
print(my_mat_char)
```

```
R>      [,1] [,2] [,3]
R> [1,] "a"  "a"  "a"
R> [2,] "b"  "b"  "b"
R> [3,] "c"  "c"  "c"
```

Essa flexibilidade dos objetos matriciais possibilita a fácil representação e visualização de seus dados em casos específicos.

### 6.3.1 Selecionando Valores de uma Matriz

Tal como no caso dos vetores atômicos, também é possível selecionar pedaços de uma matriz através de indexação. Uma diferença aqui é que os objetos de matrizes possuem duas dimensões, enquanto vetores possuem apenas uma.<sup>1</sup> Essa dimensão extra requer a indexação não apenas de linhas mas também de colunas. Os elementos de uma matriz podem ser acessados pela notação `[i,j]`, onde *i* representa a linha e *j* a coluna. Veja o exemplo a seguir:

```
my_mat <- matrix(1:9, nrow = 3)
print(my_mat)
```

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

<sup>1</sup>Para evitar confusão, vale salientar que, no R, vetores atômicos não possuem dimensão no sentido estrito da função. Ao usar a função `dim` em um vetor atômico, tal como em `dim(c(1,2,4))`, o resultado é nulo (`NULL`). Fora do ambiente computacional, porém, vetores atômicos podem ser entendidos como unidimensionais pois se expandem em apenas um sentido.

```
print(my_mat[1,2])
```

```
R> [1] 4
```

Para selecionar colunas ou linhas inteiras, basta deixar o índice vazio, tal como no exemplo a seguir:

```
print(my_mat[ , 2])
```

```
R> [1] 4 5 6
```

```
print(my_mat[1, ])
```

```
R> [1] 1 4 7
```

Observe que a indexação retorna um vetor atômico da classe dos dados. Caso quiséssemos que o pedaço da matriz mantivesse a sua classe e orientação vertical ou horizontal, poderíamos forçar essa conversão pelo uso de `matrix`:

```
print(as.matrix(my_mat[ ,2]))
```

```
R>      [,1]
```

```
R> [1,]    4
```

```
R> [2,]    5
```

```
R> [3,]    6
```

```
print(matrix(my_mat[1, ], nrow=1))
```

```
R>      [,1] [,2] [,3]
```

```
R> [1,]    1    4    7
```

Pedaços da matriz também podem ser selecionados via indexadores. Caso quiséssemos uma matriz formada a partir da seleção dos elementos da segunda linha e primeira coluna até a terceira linha e segunda coluna, usaríamos o seguinte código:

```
print(my_mat[2:3,1:2])
```

```
R>      [,1] [,2]
```

```
R> [1,]    2    5
```

```
R> [2,]    3    6
```

Por fim, o uso de testes lógicos para selecionar valores de matrizes também é possível. Veja a seguir:

```
my_mat <- matrix(1:9, nrow = 3)
print(my_mat >5)
```

```
R>      [,1] [,2] [,3]
```

```
R> [1,] FALSE FALSE TRUE
```

```
R> [2,] FALSE FALSE TRUE
```

```
R> [3,] FALSE  TRUE TRUE
```

```
print(my_mat[my_mat >5])
```

```
R> [1] 6 7 8 9
```

### 6.3.2 Outras Funções Úteis

**as.matrix** - Transforma dados para a classe `matrix`.

```
my_mat <- as.matrix(1:5)
print(my_mat)
```

```
R>      [,1]
```

```
R> [1,]    1
```

```
R> [2,]    2
```

```
R> [3,]    3
```

```
R> [4,]    4
```

```
R> [5,]    5
```

**t** - Retorna a transposta da matriz.

```
my_mat <- matrix(seq(10,20, length.out = 6), nrow = 3)
print(my_mat)
```

```
R>      [,1] [,2]
```

```
R> [1,]   10   16
```

```
R> [2,]   12   18
```

```
R> [3,]   14   20
```

```
print(t(my_mat))
```

```
R>      [,1] [,2] [,3]
```

```
R> [1,]   10   12   14
```

```
R> [2,]   16   18   20
```

**rbind** - Retorna a junção (cola) vertical de matrizes, orientando-se pelas linhas.

```
my_mat_1 <- matrix(1:5, nrow = 1)
print(my_mat_1)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
```

```
R> [1,]    1    2    3    4    5
```

```
my_mat_2 <- matrix(10:14, nrow = 1)
print(my_mat_2)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
R> [1,]  10  11  12  13  14
```

```
my.rbind.mat <- rbind(my_mat_1, my_mat_2)
print(my.rbind.mat)
```

```
R>      [,1] [,2] [,3] [,4] [,5]
R> [1,]    1    2    3    4    5
R> [2,]  10   11   12   13   14
```

**cbind** - Retorna a junção (cola) horizontal de matrizes, orientando-se pelas colunas.

```
my_mat_1 <- matrix(1:4, nrow = 2)
print(my_mat_1)
```

```
R>      [,1] [,2]
R> [1,]    1    3
R> [2,]    2    4
```

```
my_mat_2 <- matrix(10:13, nrow = 2)
print(my_mat_2)
```

```
R>      [,1] [,2]
R> [1,]  10  12
R> [2,]  11  13
```

```
my_cbind_mat <- cbind(my_mat_1, my_mat_2)
print(my_cbind_mat)
```

```
R>      [,1] [,2] [,3] [,4]
R> [1,]    1    3   10   12
R> [2,]    2    4   11   13
```

## 6.4 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.



# Capítulo 7

## As Classes Básicas de Objetos

**As classes básicas são os elementos mais primários na representação de dados no R.** Nos capítulos anteriores utilizamos as classes básicas como colunas em uma tabela. Dados numéricos viraram uma coluna do tipo `numeric`, enquanto dados de texto viraram um objeto do tipo `character`.

Neste capítulo iremos estudar mais a fundo as classes básicas de objetos do R, incluindo a sua criação até a manipulação do seu conteúdo. Este capítulo é de suma importância pois mostrará quais operações são possíveis com cada classe de objeto e como podemos manipular as informações de forma eficiente. Os tipos de objetos tratados aqui serão:

- Numéricos (`numeric`)
- Texto (`character`)
- Fatores (`factor`)
- Valores lógicos (`logical`)
- Datas e tempo (`Date` e `timedate`)
- Dados Omissos (`NA`)

### 7.1 Objetos Numéricos

Uma das classes mais utilizadas em pesquisa empírica. Os valores numéricos são representações de uma quantidade. Por exemplo: o preço de uma ação em determinada data, o volume negociado de um contrato financeiro em determinado dia, a inflação anual de um país, entre várias outras possibilidades.

### 7.1.1 Criando e Manipulando Vetores Numéricos

A criação e manipulação de valores numéricos é fácil e direta. Os símbolos de operações matemáticas seguem o esperado, tal como soma (+), diminuição (-), divisão (/) e multiplicação (\*). Todas as operações matemáticas são efetuadas com a orientação de elemento para elemento e possuem notação vetorial. Isso significa, por exemplo, que podemos manipular vetores inteiros em uma única linha de comando. Veja a seguir, onde se cria dois vetores e realiza-se diversas operações entre eles.

```
# create numeric vectors
x <- 1:5
y <- 2:6

# print sum
print(x+y)
```

```
R> [1] 3 5 7 9 11
```

```
# print multiplication
print(x*y)
```

```
R> [1] 2 6 12 20 30
```

```
# print division
print(x/y)
```

```
R> [1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
# print exponentiation
print(x^y)
```

```
R> [1] 1 8 81 1024 15625
```

Um diferencial do R em relação a outras linguagens é que, nele, são aceitas operações entre vetores diferentes. Por exemplo, podemos somar um vetor numérico de quatro elementos com outro de apenas dois. Nesse caso, aplica-se a chamada **regra de reciclagem** (*recycling rule*). Ela define que, se dois vetores de tamanho diferente estão interagindo, o vetor menor é repetido tantas vezes quantas forem necessárias para obter-se o mesmo número de elementos do vetor maior. Veja o exemplo a seguir:

```
# set x with 4 elements and y with 2
x <- 1:4
y <- 2:1

# print sum
print(x + y)
```

```
R> [1] 3 3 5 5
```

O resultado de  $x + y$  é equivalente a  $1:4 + c(2, 1, 2, 1)$ . Caso interagirmos vetores em que o tamanho do maior não é múltiplo do menor, o R realiza o mesmo procedimento de reciclagem, porém emite uma mensagem de `warning`:

```
# set x = 4 elements and y with 3
x <- c(1, 2, 3, 4)
y <- c(1, 2, 3)

# print sum (recycling rule)
print(x + y)
```

```
R> Warning in x + y: longer object length is not a multiple of
R> shorter object length
```

```
R> [1] 2 4 6 5
```

Os três primeiros elementos de  $x$  foram somados aos três primeiros elementos de  $y$ . O quarto elemento de  $x$  foi somado ao primeiro elemento de  $y$ . Uma vez que não havia um quarto elemento em  $y$ , o ciclo reinicia, resgatando o primeiro elemento de  $y$  e resultando em uma soma igual a 5.

Os elementos de um vetor numérico também podem ser nomeados quando na criação do vetor:

```
# create named vector
x <- c(item1 = 10,
       item2 = 14,
       item3 = 9,
       item4 = 2)

# print it
print(x)
```

```
R> item1 item2 item3 item4
R>   10   14    9    2
```

Para nomear os elementos após a criação, podemos utilizar a função `names`. Veja a seguir:

```
# create unnamed vector
x <- c(10, 14, 9, 2)

# set names of elements
names(x) <- c('item1', 'item2', 'item3', 'item4')
```

```
# print it
print(x)
```

```
R> item1 item2 item3 item4
R> 10 14 9 2
```

Vetores numéricos vazios também podem ser criados. Em algumas situações de desenvolvimento de código faz sentido pré-alocar o vetor antes de preenchê-lo com valores. Nesse caso, utilize a função `numeric`:

```
# create empty numeric vector of length 10
my_x <- numeric(length = 10)

# print it
print(my_x)
```

```
R> [1] 0 0 0 0 0 0 0 0 0 0
```

Observe que, nesse caso, os valores de `my_x` são definidos como zero.

#### 7.1.1.1 Criando Sequências de Valores

Existem duas maneiras de criar uma sequência de valores no R. A primeira, que já foi utilizada nos exemplos anteriores, é o uso do operador `:`. Por exemplo, `my_seq <- 1:10` ou `my_seq <- -5:5`. Esse método é bastante prático, pois a notação é clara e direta.

Porém, o uso do operador `:` limita as possibilidades. A diferença entre os valores adjacentes é sempre `1` para sequências ascendentes e `-1` para sequências descendentes. Uma versão mais poderosa para a criação de sequências é o uso da função `seq`. Com ela, é possível definir os intervalos entre cada valor com o argumento `by`. Veja a seguir:

```
# set sequence
my_seq <- seq(from = -10, to = 10, by = 2)

# print it
print(my_seq)
```

```
R> [1] -10 -8 -6 -4 -2 0 2 4 6 8 10
```

Outro atributo interessante da função `seq` é a possibilidade de criar vetores com um valor inicial, um valor final e o número de elementos desejado. Isso é realizado com o uso da opção `length.out`. Observe o código a seguir, onde cria-se um vetor de `0` até `10` com `20` elementos:

```
# set sequence with fixed size
my_seq <- seq(from = 0, to = 10, length.out = 20)

# print it
print(my_seq)
```

```
R> [1] 0.0000000 0.5263158 1.0526316 1.5789474 2.1052632
R> [6] 2.6315789 3.1578947 3.6842105 4.2105263 4.7368421
R> [11] 5.2631579 5.7894737 6.3157895 6.8421053 7.3684211
R> [16] 7.8947368 8.4210526 8.9473684 9.4736842 10.0000000
```

No caso anterior, o tamanho final do vetor foi definido e a própria função se encarregou de descobrir qual a variação necessária entre cada valor de `my_seq`.

### 7.1.1.2 Criando Vetores com Elementos Repetidos

Outra função interessante é a que cria vetores com o uso de repetição. Por exemplo: imagine que estamos interessado em um vetor preenchido com o valor 1 dez vezes. Para isso, basta utilizar a função `rep`:

```
# repeat vector three times
my_x <- rep(x = 1, times = 10)

# print it
print(my_x)
```

```
R> [1] 1 1 1 1 1 1 1 1 1 1
```

A função também funciona com vetores. Considere uma situação onde temos um vetor com os valores `c(1,2)` e gostaríamos de criar um vetor maior com os elementos `c(1, 2, 1, 2, 1, 2)` - isto é, repetindo o vetor menor três vezes. Veja o resultado a seguir:

```
# repeat vector three times
my_x <- rep(x = c(1, 2), times = 3)

# print it
print(my_x)
```

```
R> [1] 1 2 1 2 1 2
```

### 7.1.1.3 Criando Vetores com Números Aleatórios

Em muitas situações será necessário a criação de números aleatórios. Esse procedimento numérico é bastante utilizado para simular modelos matemáticos em Finanças.

Por exemplo, o método de simulação de preços de ativos de Monte Carlo parte da simulação de números aleatórios (McLeish, 2011). No R, existem diversas funções que criam números aleatórios para diferentes distribuições estatísticas. As mais utilizadas, porém, são as funções `rnorm` e `runif`.

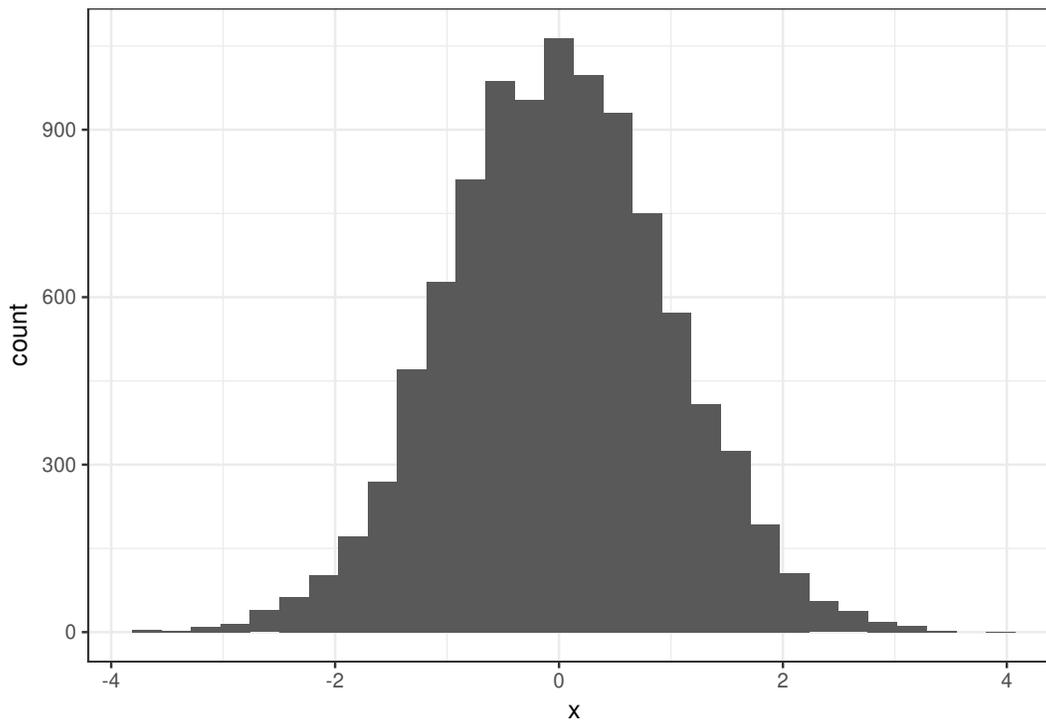
A função `rnorm` gera números aleatórios da distribuição Normal, com opções para a média (tendência) e o desvio padrão (variabilidade). Veja o seu uso a seguir:

```
# generate 10000 random numbers from a Normal distribution
my_rnd_vec <- rnorm(n = 10000,
                   mean = 0,
                   sd = 1)

# print first 20 elements
print(my_rnd_vec[1:20])
```

```
R> [1]  1.1194299  1.0155929 -1.2027493  0.5309199 -1.0571733
R> [6] -1.1854921 -0.4107645  1.5315372 -1.9588221  1.0565205
R> [11] -0.6458420 -1.2922453  1.0599906 -1.4601082  1.5176087
R> [16]  0.1781568  0.3274256  1.4443136  1.4512198  0.2217988
```

O código anterior gera uma grande quantidade de números aleatórios de uma distribuição Normal com média zero e desvio padrão igual a um. Podemos verificar a distribuição dos números gerados com um histograma:



Como esperado, temos o formato de sino que caracteriza uma distribuição do tipo Normal. Como exercício, podes trocar as entradas `mean` e `sd` e confirmar como isso afeta a figura gerada.

Já a função `runif` gera valores aleatórios da distribuição uniforme entre um valor máximo e um valor mínimo. Ela é geralmente utilizada para simular probabilidades. A função `runif` tem três parâmetros de entrada: o número de valores aleatórios desejado, o valor mínimo e o valor máximo. Veja exemplo a seguir:

```
# create a random vector with minimum and maximum
my_rnd_vec <- runif(n = 10,
                   min = -5,
                   max = 5)

# print it
print(my_rnd_vec)
```

```
R> [1] -1.203942 -2.608231 -2.733635  4.457898 -1.748036
R> [6] -4.378253  4.576698  2.310536 -2.136151  3.209011
```

Observe que ambas as funções anteriores são limitadas à sua respectiva distribuição. Uma maneira alternativa e flexível de gerar valores aleatórios é utilizar a função `sample`. Essa tem como entrada um vetor qualquer e retorna uma versão embaralhada de seus elementos. A sua flexibilidade reside no fato de que o vetor de entrada

pode ser qualquer coisa. Por exemplo, caso quiséssemos criar um vetor aleatório com os números `c(0, 5, 15, 20, 25)` apenas, poderíamos fazê-lo da seguinte forma:

```
# create sequence
my_vec <- seq(from = 0, to = 25, by=5)

# sample sequence
my_rnd_vec <- sample(my_vec)

# print it
print(my_rnd_vec)
```

```
R> [1] 15  0 20 25  5 10
```

A função `sample` também permite a seleção aleatória de um certo número de termos. Por exemplo, caso quiséssemos selecionar aleatoriamente apenas um elemento de `my_vec`, escreveríamos o código da seguinte maneira:

```
# sample one element of my_vec
my_rnd_vec <- sample(my_vec, size = 1)

# print it
print(my_rnd_vec)
```

```
R> [1] 5
```

Caso quiséssemos dois elementos, escreveríamos:

```
# sample two elements of my_vec
my_rnd_vec <- sample(my_vec, size = 2)

# print it
print(my_rnd_vec)
```

```
R> [1] 25  0
```

Também é possível selecionar valores de uma amostra menor para a criação de um vetor maior. Por exemplo, considere o caso em que se tem um vetor com os números `c(10, 15, 20)` e deseja-se criar um vetor aleatório com dez elementos retirados desse vetor menor, com repetição. Para isso, podemos utilizar a opção `replace = TRUE`.

```
# create vector
my_vec <- c(5, 10, 15)

# sample
my_rnd_vec <- sample(x = my_vec, size = 10, replace = TRUE)
```

```
print(my_rnd_vec)
```

```
R> [1] 15 10 10 5 5 10 5 5 15 10
```

Vale destacar que a função `sample` funciona para qualquer tipo ou objeto, não sendo, portanto, exclusiva para vetores numéricos. Poderíamos, também, escolher elementos aleatórios de um vetor de texto ou então uma lista:

```
# example of sample with characters
print(sample(c('elem 1', 'elem 2', 'elem 3'),
            size = 1))
```

```
R> [1] "elem 3"
```

```
# example of sample with list
print(sample(list(x = c(1,1,1),
                    y = c('a', 'b')),
            size = 1))
```

```
R> $x
```

```
R> [1] 1 1 1
```

É importante ressaltar que a geração de valores aleatórios no R (ou qualquer outro programa) **não é totalmente aleatória!** De fato, o próprio computador escolhe os valores dentre uma fila de valores possíveis. Cada vez que funções tal como `rnorm`, `runif` e `sample` são utilizadas, o computador escolhe um lugar diferente dessa fila de acordo com vários parâmetros, incluindo a data e horário atual. Portanto, do ponto de vista do usuário, os valores são gerados de forma imprevisível. Para o computador, porém, essa seleção é completamente determinística e guiada por regras claras.

Uma propriedade interessante no R é a possibilidade de selecionar uma posição específica na fila de valores aleatórios utilizando função `set.seed`. É ela que **fixa a semente** para gerar os valores. Na prática, o resultado é que todos os números e seleções aleatórias realizadas pelo código serão iguais em cada execução. O uso de `set.seed` é bastante recomendado para manter a reprodutibilidade dos códigos envolvendo aleatoriedade. Veja o exemplo a seguir, onde utiliza-se essa função.

```
# fix seed
set.seed(seed = 10)

# set vec and print
my_rnd_vec_1 <- runif(5)
print(my_rnd_vec_1)
```

```
R> [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
```

```
# set vec and print
my_rnd_vec_2 <- runif(5)
print(my_rnd_vec_2)
```

```
R> [1] 0.2254366 0.2745305 0.2723051 0.6158293 0.4296715
```

No código anterior, o valor de `set.seed` é um inteiro escolhido pelo usuário. Após a chamada de `set.seed`, todas as seleções e números aleatórios irão iniciar do mesmo ponto e, portanto, serão iguais. Motivo o leitor a executar o código anterior no R. Verás que os valores de `my_rnd_vec_1` e `my_rnd_vec_2` serão exatamente iguais aos valores colocados aqui.

O uso de `set.seed` também funciona para o caso de `sample`. Veja a seguir:

```
# fix seed
set.seed(seed = 15)

# print vectors
print(sample(1:10))
```

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

```
print(sample(10:20))
```

```
R> [1] 13 15 10 17 20 14 19 12 11 18 16
```

Novamente, execute os comandos anteriores no R e verás que o resultado na tela bate com o apresentado aqui.

### 7.1.2 Acessando Elementos de um Vetor Numérico

Todos os elementos de um vetor numérico podem ser acessados através do uso de colchetes (`[ ]`). Por exemplo, caso quiséssemos apenas o primeiro elemento de `x`, teríamos:

```
# set vector
x <- c(-1, 4, -9, 2)

# get first element
first_elem_x <- x[1]

# print it
print(first_elem_x)
```

```
R> [1] -1
```

A mesma notação é válida para extrair porções de um vetor. Caso quiséssemos um subvetor de `x` com o primeiro e o segundo elemento, faríamos essa operação da seguinte forma:

```
# sub-vector of x
sub_x <- x[1:2]

# print it
print(sub_x)
```

```
R> [1] -1 4
```

Para acessar elementos nomeados de um vetor numérico, basta utilizar seu nome junto aos colchetes.

```
# set named vector
x <- c(item1 = 10, item2 = 14, item3 = -9, item4 = -2)

# access elements by name
print(x['item2'])
```

```
R> item2
```

```
R> 14
```

```
print(x[c('item2', 'item4')])
```

```
R> item2 item4
```

```
R> 14 -2
```

O acesso aos elementos de um vetor numérico também é possível através de testes lógicos. Por exemplo, caso tivéssemos interesse em saber quais os valores de `x` que são maiores do que 0, o código resultante seria da seguinte forma:

```
# find all values of x higher than zero
print(x[x > 0])
```

```
R> item1 item2
```

```
R> 10 14
```

Os usos de regras de segmentação dos dados de acordo com algum critério é chamado de indexação lógica. Os objetos do tipo `logical` serão tratados mais profundamente em seção futura deste capítulo.

### 7.1.3 Modificando e Removendo Elementos de um Vetor Numérico

A modificação de um vetor numérico é muito simples. Basta indicar a posição dos elementos e os novos valores com o símbolo de *assign* (`<-`):

```
# set vector
my_x <- 1:4

# modify first element to 5
my_x[1] <- 5

# print result
print(my_x)
```

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

Essa modificação também pode ser realizada em bloco:

```
# set vector
my_x <- 0:5

# set the first three elements to 5
my_x[1:3] <- 5

# print result
print(my_x)
```

```
R> [1] 5 5 5 3 4 5
```

O uso de condições para definir elementos é realizada pela indexação:

```
# set vector
my_x <- -5:5

# set any value lower than 2 to 0
my_x[my_x<2] <- 0

# print result
print(my_x)
```

```
R> [1] 0 0 0 0 0 0 2 3 4 5
```

A remoção de elementos é realizada com o uso de índices negativos:

```
# create vector
my_x <- -5:5

# remove first and second element of my_x
my_x <- my_x[-(1:2)]

# show result
```

```
print(my_x)
```

```
R> [1] -3 -2 -1  0  1  2  3  4  5
```

Note como o uso do índice negativo em `my_x[-(1:2)]` retorna o vetor original sem o primeiro e segundo elemento.

#### 7.1.4 Criando Grupos

Em algumas situações será necessário entender quantos casos da amostra estão localizados entre um determinado intervalo. Por exemplo, imagine o vetor dos retornos diários de uma ação, isto é, a variação percentual dos preços de fechamento entre um dia e outro. Uma possível análise de risco que pode ser realizada é dividir o intervalo de retornos em cinco partes e verificar o percentual de ocorrência dos valores em cada um dos intervalos. Essa análise numérica é bastante semelhante à construção e visualização de histogramas.

A função `cut` serve para criar grupos de intervalos a partir de um vetor numérico. Veja o exemplo a seguir, onde cria-se um vetor aleatório oriundo da distribuição Normal e cinco grupos a partir de intervalos definidos pelos dados.

```
# set rnd vec
my_x <- rnorm(10)

# "cut" it into 5 pieces
my_cut <- cut(x = my_x, breaks = 5)
print(my_cut)
```

```
R> [1] (-1.57,-1.12] (0.252,0.71] (-1.12,-0.66]
R> [4] (-0.204,0.252] (-0.66,-0.204] (-1.57,-1.12]
R> [7] (0.252,0.71] (-0.204,0.252] (0.252,0.71]
R> [10] (-0.204,0.252]
R> 5 Levels: (-1.57,-1.12] (-1.12,-0.66] ... (0.252,0.71]
```

Observe que os nomes dos elementos da variável `my_cut` são definidos pelos intervalos e o resultado é um objeto do tipo fator. Em seções futuras, iremos explicar melhor esse tipo de objeto e as suas propriedades.

No exemplo anterior, os intervalos para cada grupo foram definidos automaticamente. No uso da função `cut`, também é possível definir quebras customizadas nos dados e nos nomes dos grupos. Veja a seguir:

```
# set random vector
my_x <- rnorm(10)

# create groups with 5 breaks
```

```
my_cut <- cut(x = my_x, breaks = 5)
```

```
# print it!
print(my_cut)
```

```
R> [1] (-1.3,-0.3] (-0.3,0.697] (-0.3,0.697] (-2.3,-1.3]
R> [5] (-0.3,0.697] (0.697,1.69] (0.697,1.69] (0.697,1.69]
R> [9] (-1.3,-0.3] (1.69,2.7]
R> 5 Levels: (-2.3,-1.3] (-1.3,-0.3] ... (1.69,2.7]
```

Note que os nomes dos elementos em `my_cut` foram definidos como intervalos e o resultado é um objeto do tipo fator. É possível também definir intervalos e nomes customizados para cada grupo com o uso dos argumentos `labels` e `breaks`:

```
# create random vector
```

```
my_x <- rnorm(10)
```

```
# define breaks manually
```

```
my_breaks <- c(min(my_x)-1, -1, 1, max(my_x)+1)
```

```
# define labels manually
```

```
my_labels <- c('Low', 'Normal', 'High')
```

```
# create group from numerical vector
```

```
my_cut <- cut(x = my_x, breaks = my_breaks, labels = my_labels)
```

```
# print both!
```

```
print(my_x)
```

```
R> [1] 0.5981759 1.6113647 -0.4373813 1.3526206 0.4705685
R> [6] 0.4702481 0.3963088 -0.7304926 0.6531176 1.2279598
```

```
print(my_cut)
```

```
R> [1] Normal High Normal High Normal Normal Normal Normal
```

```
R> [9] Normal High
```

```
R> Levels: Low Normal High
```

Como podemos ver, os nomes dos grupos estão mais amigáveis para uma futura análise.

### 7.1.5 Outras Funções Úteis

**as.numeric** - Converte determinado objeto para numérico.

```
my_text <- c('1', '2', '3')
class(my_text)
```

```
R> [1] "character"
```

```
my_x <- as.numeric(my_text)
print(my_x)
```

```
R> [1] 1 2 3
```

```
class(my_x)
```

```
R> [1] "numeric"
```

**sum** - Soma os elementos de um vetor.

```
my_x <- 1:50
my_sum <- sum(my_x)
print(my_sum)
```

```
R> [1] 1275
```

**max** - Retorna o máximo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
max_x <- max(x)
print(max_x)
```

```
R> [1] 14
```

**min** - Retorna o mínimo valor numérico do vetor.

```
x <- c(12, 15, 9, 2)
min_x <- min(x)
print(min_x)
```

```
R> [1] 2
```

**which.max** - Retorna a posição do máximo valor numérico do vetor.

```
x <- c(100, 141, 9, 2)
which.max_x <- which.max(x)
cat(paste('The position of the maximum value of x is ', which.max_x))
```

```
R> The position of the maximum value of x is 2
```

```
cat(' and its value is ', x[which.max_x])
```

```
R> and its value is 141
```

**which.min** - Retorna a posição do mínimo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
which.min_x <- which.min(x)
cat(paste('The position of the minimum value of x is ',
         which.min_x, ' and its value is ', x[which.min_x]))
```

```
R> The position of the minimum value of x is 4 and its value is 2
```

**sort** - Retorna uma versão ordenada de um vetor.

```
x <- runif(5)
print(sort(x, decreasing = FALSE))
```

```
R> [1] 0.1623069 0.8347800 0.8553657 0.9099027 0.9935257
```

```
print(sort(x, decreasing = TRUE))
```

```
R> [1] 0.9935257 0.9099027 0.8553657 0.8347800 0.1623069
```

**cumsum** - Soma os elementos de um vetor de forma cumulativa.

```
my_x <- 1:25
my_cumsum <- cumsum(my_x)
print(my_cumsum)
```

```
R> [1] 1 3 6 10 15 21 28 36 45 55 66 78 91 105
```

```
R> [15] 120 136 153 171 190 210 231 253 276 300 325
```

**prod** - Realiza o produto de todos os elementos de um vetor.

```
my_x <- 1:10
my_prod <- prod(my_x)
print(my_prod)
```

```
R> [1] 3628800
```

**cumprod** - Calcula o produto cumulativo de todos os elementos de um vetor.

```
my_x <- 1:10
my_prod <- cumprod(my_x)
print(my_prod)
```

```
R> [1] 1 2 6 24 120 720 5040
```

```
R> [8] 40320 362880 3628800
```

## 7.2 Classe de Caracteres (texto)

A classe de caracteres, ou texto, serve para armazenar informações textuais. Um exemplo prático em Finanças seria o reconhecimento de uma ação através dos seus símbolos de identificação (*tickers*) ou então por sua classe de ação: ordinária ou preferencial. Este tipo de dado tem sido utilizado cada vez mais em pesquisa empírica (Gentzkow et al., 2017), resultando em uma diversidade de pacotes.

O R possui vários recursos que facilitam a criação e manipulação de objetos de tipo texto. As funções básicas fornecidas com a instalação de R são abrangentes e adequadas para a maioria dos casos. No entanto, pacote `stringr` (Wickham, 2022b) do `tidyverse` fornece muitas funções que expandem a funcionalidade básica do R na manipulação de texto.

Um aspecto positivo de `stringr` é que as funções começam com o nome `str_` e possuem nomes informativos. Combinando isso com o recurso de preenchimento automático (*autocomplete*) pela tecla `tab`, fica fácil de localizar os nomes das funções do pacote. Seguindo a prioridade ao universo do `tidyverse`, esta seção irá dar preferência ao uso das funções do pacote `stringr`. As rotinas nativas de manipulação de texto serão apresentadas, porém de forma limitada.

### 7.2.1 Criando um Objeto Simples de Caracteres

Todo objeto de caracteres é criado através da encapsulação de um texto por aspas duplas (" ") ou simples (' '). Para criar um vetor de caracteres com *tickers* de ações, podemos fazê-lo com o seguinte código:

```
my_assets <- c('PETR3', 'VALE4', 'GGBR4')
print(my_assets)
```

```
R> [1] "PETR3" "VALE4" "GGBR4"
```

Confirma-se a classe do objeto com a função `class`:

```
class(my_assets)
```

```
R> [1] "character"
```

### 7.2.2 Criando Objetos Estruturados de Texto

Em muitos casos no uso do R, estaremos interessados em criar vetores de texto com algum tipo de estrutura própria. Por exemplo, o vetor `c("text 1", "text 2", ..., "text 20")` possui uma lógica de criação clara. Computacionalmente, podemos definir a sua estrutura como sendo a junção do texto `text` e um vetor de sequência, de 1 até 20.

Para criar um vetor textual capaz de unir texto com número, utilizamos a função `stringr::str_c` ou `base::paste`. Veja o exemplo a seguir, onde replica-se o caso anterior com e sem espaço entre número e texto:

```
library(stringr)
```

```
# create sequence
```

```
my_seq <- 1:20
```

```
# create character
```

```
my_text <- 'text'
```

```
# paste objects together (without space)
```

```
my_char <- str_c(my_text, my_seq)
```

```
print(my_char)
```

```
R> [1] "text1" "text2" "text3" "text4" "text5" "text6"
R> [7] "text7" "text8" "text9" "text10" "text11" "text12"
R> [13] "text13" "text14" "text15" "text16" "text17" "text18"
R> [19] "text19" "text20"
```

```
# paste objects together (with space)
```

```
my_char <- str_c(my_text, my_seq, sep = ' ')
```

```
print(my_char)
```

```
R> [1] "text 1" "text 2" "text 3" "text 4" "text 5"
R> [6] "text 6" "text 7" "text 8" "text 9" "text 10"
R> [11] "text 11" "text 12" "text 13" "text 14" "text 15"
R> [16] "text 16" "text 17" "text 18" "text 19" "text 20"
```

```
# paste objects together (with space)
```

```
my_char <- paste(my_text, my_seq)
```

```
print(my_char)
```

```
R> [1] "text 1" "text 2" "text 3" "text 4" "text 5"
R> [6] "text 6" "text 7" "text 8" "text 9" "text 10"
R> [11] "text 11" "text 12" "text 13" "text 14" "text 15"
R> [16] "text 16" "text 17" "text 18" "text 19" "text 20"
```

O mesmo procedimento também pode ser realizado com vetores de texto. Veja a seguir:

```
# set character value
```

```
my_x <- 'My name is'
```

```
# set character vector
my_names <- c('Marcelo', 'Ricardo', 'Tarcizio')

# paste and print
print(str_c(my_x, my_names, sep = ' '))
```

```
R> [1] "My name is Marcelo" "My name is Ricardo"
R> [3] "My name is Tarcizio"
```

Outra possibilidade de construção de textos estruturados é a repetição do conteúdo de um objeto do tipo caractere. No caso de texto, utiliza-se a função `stringr::str_dup/base::strrep` para esse fim. Observe o exemplo a seguir:

```
my_char <- str_dup(string = 'abc', times = 5)
print(my_char)
```

```
R> [1] "abcabcabcabcabc"
```

### 7.2.3 Objetos Constantes de Texto

O R também possibilita o acesso direto a todas as letras do alfabeto. Esses estão guardadas nos objetos reservados chamados `letters` e `LETTERS`:

```
# print all letters in alphabet (no cap)
print(letters)
```

```
R> [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
R> [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
# print all letters in alphabet (WITH CAP)
print(LETTERS)
```

```
R> [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
R> [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Observe que em ambos os casos não é necessário criar os objetos. Por serem constantes embutidas automaticamente na área de trabalho do R, elas já estão disponíveis para uso. Podemos sobrescrever o nome do objeto com outro conteúdo, porém isso não é aconselhável. Nunca se sabe onde esse objeto constante está sendo usado. Outros objetos de texto constantes no R incluem `month.abb` e `month.name`. Veja a seguir o seu conteúdo:

```
# print abbreviation and full names of months
print(month.abb)
```

```
R> [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
R> [10] "Oct" "Nov" "Dec"
```

```
print(month.name)
```

```
R> [1] "January" "February" "March"    "April"
R> [5] "May"     "June"     "July"     "August"
R> [9] "September" "October"  "November" "December"
```

#### 7.2.4 Selecionando Pedacos de um Texto

Um erro comum praticado por iniciantes é tentar selecionar pedaços de um texto através do uso de colchetes. Observe o código abaixo:

```
# set char object
my_char <- 'ABCDE'

# print its second character: 'B' (WRONG - RESULT is NA)
print(my_char[2])
```

```
R> [1] NA
```

O resultado NA indica que o segundo elemento de `my_char` não existe. Isso acontece porque o uso de colchetes refere-se ao acesso de **elementos** de um vetor atômico, e não de caracteres dentro de um texto maior. Observe o que acontece quando utilizamos `my_char[1]`:

```
print(my_char[1])
```

```
R> [1] "ABCDE"
```

O resultado é simplesmente o texto `ABCDE`, que está localizado no primeiro item de `my_char`. Para selecionar pedaços de um texto, devemos utilizar a função específica `stringr::str_sub/base::substr`:

```
# print third and fourth characters
my_substr <- str_sub(string = my_char,
                    start = 4,
                    end = 4)

print(my_substr)
```

```
R> [1] "D"
```

Essa função também funciona para vetores atômicos. Vamos assumir que você importou dados de texto e o conjunto de dados bruto contém um identificador de 3 dígitos de uma empresa, sempre na mesma posição do texto. Vamos simular a situação no R:

```
# build char vec
my_char_vec <- paste0(c('123', '231', '321'),
```

```
      ' - other ignorable text')
print(my_char_vec)
```

```
R> [1] "123 - other ignorable text"
R> [2] "231 - other ignorable text"
R> [3] "321 - other ignorable text"
```

Só estamos interessados na informação das três primeiras letras de cada elemento em `my_char_vec`. Para selecioná-los, podemos usar as mesmas funções que antes.

```
# get ids with stringr::str_sub
ids.vec <- str_sub(my_char_vec, 1, 3)
print(ids.vec)
```

```
R> [1] "123" "231" "321"
```



**Operações vetorizadas são comuns e esperadas no R.** Quase tudo o que você pode fazer para um único elemento pode ser expandido para vetores. Isso facilita o desenvolvimento de rotinas pois pode-se facilmente realizar tarefas complicadas em uma série de elementos, em uma única linha de código.

### 7.2.5 Localizando e Substituindo Pedacos de um Texto

Uma operação útil na manipulação de textos é a localização de letras e padrões específicos com funções `stringr::str_locate/base::regexr` e `stringr::str_locate_all/base::gregexpr`. É importante destacar que, por *default*, essas funções utilizam de expressões do tipo *regex* - expressões regulares (Thompson, 1968). Essa é uma linguagem específica para processar textos. Diversos símbolos são utilizados para estruturar, procurar e isolar padrões textuais. Quando utilizada corretamente, o *regex* é bastante útil e de extrema valia.

Usualmente, o caso mais comum em pesquisa é verificar a posição ou a existência de um texto menor dentro de um texto maior. Isto é, um padrão explícito e fácil de entender. Por isso, a localização e substituição de caracteres no próximo exemplo será do tipo fixo, sem o uso de *regex*. Tal informação pode ser passada às funções do pacote `stringr` através de outra função chamada `stringr::fixed`.

O exemplo a seguir mostra como encontrar o caractere *D* dentre uma série de caracteres.

```
library(stringr)

my_char <- 'ABCDEF-ABCDEF-ABC'
```

```
pos = str_locate(string = my_char, pattern = fixed('D'))
print(pos)
```

```
R>      start end
R> [1,]      4  4
```

Observe que a função `str_locate` retorna apenas a primeira ocorrência de `D`. Para resgatar todas as ocorrências, devemos utilizar a função `str_locate_all`:

```
# set object
my_char <- 'ABCDEF-ABCDEF-ABC'

# find position of ALL 'D' using str_locate_all
pos = str_locate_all(string = my_char, pattern = fixed('D'))
print(pos)
```

```
R> [[1]]
R>      start end
R> [1,]      4  4
R> [2,]     11 11
```

Observe também que as funções `regexp` e `grepexpr` retornam objetos com propriedades específicas, apresentando-as na tela.

Para substituir caracteres em um texto, basta utilizar a função `stringr::str_replace` ou `base::sub` e `str_replace_all` ou `base::gsub`. Vale salientar que `str_replace` substitui a primeira ocorrência do caractere, enquanto `str_replace_all` executa uma substituição global - isto é, aplica-se a todas as ocorrências. Veja a diferença a seguir:

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with sub
my_char <- sub(x = my_char,
              pattern = 'ABC',
              replacement = 'XXX')
print(my_char)
```

```
R> [1] "XXXDEF-ABCDEF-ABC"
```

```
# substitute the FIRST 'ABC' for 'XXX' with str_replace
my_char <- 'ABCDEF-ABCDEF-ABC'
my_char <- str_replace(string = my_char,
                      pattern = fixed('ABC'),
                      replacement = 'XXX')
```

```
print(my_char)
```

```
R> [1] "XXXDEF-ABCDEF-ABC"
```

E agora fazemos uma substituição global dos caracteres.

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with str_replace
my_char <- str_replace_all(string = my_char,
                           pattern = 'ABC',
                           replacement = 'XXX')

print(my_char)
```

```
R> [1] "XXXDEF-XXXDEF-XXX"
```

Mais uma vez, vale ressaltar que as operações de substituição também funcionam em vetores. Dê uma olhada no próximo exemplo.

```
# set char object
my_char <- c('ABCDEF', 'DBCFE', 'ABC')

# create an example of vector
my_char_vec <- str_c(sample(my_char, 5, replace = T),
                    sample(my_char, 5, replace = T),
                    sep = ' - ')

# show it
print(my_char_vec)
```

```
R> [1] "ABCDEF - ABC"      "ABCDEF - ABCDEF" "ABCDEF - ABC"
R> [4] "ABCDEF - DBCFE"   "DBCFE - ABCDEF"
```

```
# substitute all occurrences of 'ABC'
my_char_vec <- str_replace_all(string = my_char_vec,
                               pattern = 'ABC',
                               replacement = 'XXX')

# print result
print(my_char_vec)
```

```
R> [1] "XXXDEF - XXX"      "XXXDEF - XXXDEF" "XXXDEF - XXX"
R> [4] "XXXDEF - DBCFE"   "DBCFE - XXXDEF"
```

### 7.2.6 Separando Textos

Em algumas situações, principalmente no processamento de textos, é possível que se esteja interessado em quebrar um texto de acordo com algum separador. Por exemplo, o texto `abc ; bcd ; adf` apresenta informações demarcadas pelo símbolo `;`. Para separar um texto em várias partes, utilizamos a função `stringr::str_split/base::strsplit`. Essas quebram o texto em diversas partes de acordo com algum caractere escolhido. Observe os exemplos a seguir:

```
# set char
my_char <- 'ABCXABCXBCD'

# split it based on 'X' and using stringr::str_split
split_char <- str_split(my_char, 'X')

# print result
print(split_char)
```

```
R> [[1]]
R> [1] "ABC" "ABC" "BCD"
```

A saída dessa função é um objeto do tipo lista. Para acessar os elementos de uma lista, deve-se utilizar o operador `[[ ]]`. Por exemplo, para acessar o texto `bcd` da lista `split_char`, executa-se o seguinte código:

```
print(split_char[[1]][2])
```

```
R> [1] "ABC"
```

Para visualizar um exemplo de dividir textos em vetores, veja o próximo código.

```
# set char
my_char_vec <- c('ABCDEF', 'DBCFE', 'ABFC', 'ACD')

# split it based on 'B' and using stringr::strsplit
split_char <- str_split(my_char_vec, 'B')

# print result
print(split_char)
```

```
R> [[1]]
R> [1] "A"      "CDEF"
R>
R> [[2]]
R> [1] "D"      "CFE"
R>
```

```
R> [[3]]
R> [1] "A"  "FC"
R>
R> [[4]]
R> [1] "ACD"
```

Observe como, novamente, um objeto do tipo `list` é retornado. Cada elemento é correspondente ao processo de quebra de texto em `my_char`.

### 7.2.7 Descobrimo o Número de Caracteres de um Texto

Para descobrir o número de caracteres de um texto, utilizamos a função `stringr::str_length/base::nchar`. Ela também funciona para vetores atômicos de texto. Veja os exemplos mostrados a seguir:

```
# set char
my_char <- 'abcdef'

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 6
```

E agora um exemplo com vetores.

```
#set char
my_char <- c('a', 'ab', 'abc')

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 1 2 3
```

### 7.2.8 Gerando Combinações de Texto

Um truque útil no R é usar as funções `base::outer` e `base::expand.grid` para criar todas as combinações possíveis de elementos em diferentes objetos. Isso é útil quando você quer criar um vetor de texto combinando todos os elementos possíveis de diferentes vetores. Por exemplo, se quisermos criar um vetor com todas as combinações entre `c('a', 'b')` e `c('A', 'B')` como `c('a-A', 'a-B', ...)`, podemos escrever:

```
# set char vecs
my_vec_1 <- c('a', 'b')
my_vec_2 <- c('A', 'B')
```

```
# combine in matrix
comb.mat <- outer(my_vec_1,
                 my_vec_2,
                 paste, sep = '-')

# print it!
print(comb.mat)
```

```
R>      [,1] [,2]
R> [1,] "a-A" "a-B"
R> [2,] "b-A" "b-B"
```

A saída de `outer` é um objeto do tipo matriz. Se quisermos mudar `comb.mat` para um vetor atômico, podemos usar a função `as.character`:

```
print(as.character(comb.mat))
```

```
R> [1] "a-A" "b-A" "a-B" "b-B"
```

Outra maneira de atingir o mesmo objetivo é usar a função `expand.grid`. Veja o próximo exemplo.

```
library(tidyverse)

# set vectors
my_vec_1 <- c('John ', 'Claire ', 'Adam ')
my_vec_2 <- c('is fishing.', 'is working.')
```

```
# create df with all combinations
my_df <- expand.grid(name = my_vec_1,
                    verb = my_vec_2)

# print df
print(my_df)
```

```
R>   name      verb
R> 1  John is fishing.
R> 2 Claire is fishing.
R> 3  Adam is fishing.
R> 4  John is working.
R> 5 Claire is working.
R> 6  Adam is working.
```

```
# paste columns together in tibble
my_df <- my_df %>%
```

```
mutate(phrase = paste0(name, verb) )

# print result
print(my_df)
```

```
R>      name      verb      phrase
R> 1  John is fishing. John is fishing.
R> 2  Claire is fishing. Claire is fishing.
R> 3   Adam is fishing. Adam is fishing.
R> 4   John is working. John is working.
R> 5  Claire is working. Claire is working.
R> 6   Adam is working. Adam is working.
```

Aqui, usamos a função `expand.grid` para criar um `dataframe` contendo todas as combinações possíveis de `my_vec_1` e `my_vec_2`. Posteriormente, colamos o conteúdo das colunas do `dataframe` usando `str_c`.

### 7.2.9 Codificação de Objetos `character`

Para o R, um *string* de texto é apenas uma sequência de *bytes*. A tradução de *bytes* para caracteres é realizada de acordo com uma estrutura de codificação. Para a maioria dos casos de uso do R, especialmente em países de língua inglesa, a codificação de caracteres não é um problema pois os textos importados no R já possuem a codificação correta. Ao lidar com dados de texto em diferentes idiomas, tal como Português do Brasil, a codificação de caracteres é algo que você deve entender pois eventualmente precisará lidar com isso.

Vamos explorar um exemplo. Aqui, vamos importar dados de um arquivo de texto com a codificação 'ISO-8859-9' e verificar o resultado.

```
# read text file
my_f <- adfeR::get_data_file('FileWithLatinChar_Latin1.txt')

my_char <- readr::read_lines(my_f)

# print it
print(my_char)
```

```
R> [1] "A casa \xe9 bonita e tem muito espa\xe7o"
```

O conteúdo original do arquivo é um texto em português. Como você pode ver, a saída de `readr::read_lines` mostra todos os caracteres latinos com símbolos estranhos. Isso ocorre pois a codificação foi manualmente trocada para 'ISO-8859-9', enquanto a função `read_lines` utiliza 'UTF-8' como *default*. A solução mais fácil

e direta é modificar a codificação esperada do arquivo nas entradas de `read_lines`. Veja a seguir, onde importamos um arquivo com a codificação correta ('Latin1'):

```
my_char <- readr::read_lines(my_f,
                             locale = readr::locale(encoding='Latin1'))

# print it
print(my_char)
```

```
R> [1] "A casa é bonita e tem muito espaço"
```

Os caracteres latinos agora estão corretos pois a codificação em `read_lines` é a mesma do arquivo, 'Latin1'. Uma boa política neste tópico é sempre verificar a codificação de arquivos de texto importados e combiná-lo em R. A maioria das funções de importação tem uma opção para fazê-lo. Quando possível, sempre dê preferência para 'UTF-8'. Caso necessário, programas de edição de texto, tal como o notepad++, possuem ferramentas para verificar e trocar a codificação de um arquivo.

### 7.2.10 Outras Funções Úteis

`stringr::str_to_lower/base::tolower` - Converte um objeto de texto para letras minúsculas.

```
print(stringr::str_to_lower('ABC'))
```

```
R> [1] "abc"
```

`stringr::str_to_upper/base::toupper` - Convertem um texto em letras maiúsculas.

```
print(toupper('abc'))
```

```
R> [1] "ABC"
```

```
print(stringr::str_to_upper('abc'))
```

```
R> [1] "ABC"
```

## 7.3 Fatores

A classe de fatores (`factor`) é utilizada para representar grupos ou categorias dentro de uma base de dados no formato tabular. Por exemplo, imagine um banco de informações com os gastos de diferentes pessoas ao longo de um ano. Nessa base de dados existe um item que define o gênero do indivíduo: masculino ou feminino (M ou F). Essa respectiva coluna pode ser importada e representada como texto,

porém, no R, a melhor maneira de representá-la é através do objeto `fator`, uma vez que a mesma representa uma categoria.

A classe de fatores oferece um significado especial para denotar grupos dentro dos dados. Essa organização é integrada aos pacotes e facilita muito a vida do usuário. Por exemplo, caso quiséssemos criar um gráfico para cada grupo dentro da nossa base de dados, poderíamos fazer o mesmo simplesmente indicando a existência de uma variável de fator para a função de criação da figura. Outra possibilidade é determinar se as diferentes médias de uma variável numérica são estatisticamente diferentes para os grupos dos nossos dados. Podemos também estimar um determinado modelo estatístico para cada grupo. Quando os dados de categorias são representados apropriadamente, o uso das funções do R torna-se mais fácil e eficiente.

### 7.3.1 Criando Fatores

A criação de fatores dá-se através da função `factor`:

```
my_factor <- factor(c('M', 'F', 'M', 'M', 'F'))
print(my_factor)
```

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

Observe, no exemplo anterior, que a apresentação de fatores com a função `print` mostra os seus elementos e também o item chamado `Levels`. Esse último identifica os possíveis grupos que abrangem o vetor - nesse caso apenas M e F. Se tivéssemos um número maior de grupos, o item `Levels` aumentaria. Veja a seguir:

```
my_factor <- factor(c('M', 'F', 'M', 'M', 'F', 'ND'))
print(my_factor)
```

```
R> [1] M F M M F ND
R> Levels: F M ND
```

Um ponto importante na criação de fatores é que os `Levels` são inferidos através dos dados criados, e isso pode não corresponder à realidade. Por exemplo, observe o seguinte exemplo:

```
my_status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'))
print(my_status)
```

```
R> [1] Solteiro Solteiro Solteiro
R> Levels: Solteiro
```

Nota-se que, por ocasião, os dados mostram apenas uma categoria: `Solteiro`. Entretanto, sabe-se que outra categoria do tipo `Casado` é esperada. No caso de utilizarmos o objeto `my_status` da maneira que foi definida anteriormente, omitiremos

a informação de outros gêneros, e isso pode ocasionar problemas no futuro tal como a criação de gráficos incompletos. Nessa situação, o correto é definir os `Levels` manualmente da seguinte maneira:

```
my_status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'),
                   levels = c('Solteiro', 'Casado'))
print(my_status)
```

```
R> [1] Solteiro Solteiro Solteiro
R> Levels: Solteiro Casado
```

### 7.3.2 Modificando Fatores

Um ponto importante sobre os objetos do tipo fator é que seus `Levels` são **imutáveis e não atualizam-se com a entrada de novos dados**. Em outras palavras, não é possível modificar os valores dos `Levels` após a criação do objeto. Toda nova informação que não for compatível com os `Levels` do objeto será transformada em `NA` (*Not available*) e uma mensagem de `warning` irá aparecer na tela. Essa limitação pode parecer estranha a primeira vista porém, na prática, ela evita possíveis erros no código. Veja o exemplo a seguir:

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change first element of a factor to 'c'
my_factor[1] <- 'c'
```

```
R> Warning in `[<-factor`(`*tmp*`, 1, value = "c"): invalid
R> factor level, NA generated
```

```
# print result
print(my_factor)
```

```
R> [1] <NA> b    a    b
R> Levels: a b
```

Nesse caso, a maneira correta de proceder é primeiro transformar o objeto da classe fator para a classe caractere e depois realizar a conversão:

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change factor to character
my_char <- as.character(my_factor)

# change first element
```

```
my_char[1] <- 'c'

# mutate it back to class factor
my_factor <- factor(my_char)

# show result
print(my_factor)
```

```
R> [1] c b a b
R> Levels: a b c
```

Utilizando essas etapas temos o resultado desejado no vetor `my_factor`, com a definição de três Levels: a, b e c.

O universo `tidyverse` também possui um pacote próprio para manipular fatores, o `forcats`. Para o problema atual de modificação de fatores, podemos utilizar função `forcats::fct_recode`. Veja um exemplo a seguir, onde trocamos as siglas dos fatores:

```
library(forcats)

# set factor
my.fct <- factor(c('A', 'B', 'C', 'A', 'C', 'M', 'N'))

# modify factors
my.fct <- fct_recode(my.fct,
                     'D' = 'A',
                     'E' = 'B',
                     'F' = 'C')

# print result
print(my.fct)
```

```
R> [1] D E F D F M N
R> Levels: D E F M N
```

Observe como o uso da função `forcats::fct_recode` é intuitivo. Basta indicar o novo nome dos grupos com o operador de igualdade.

### 7.3.3 Convertendo Fatores para Outras Classes

Outro ponto importante no uso de fatores é a sua conversão para outras classes, especialmente a numérica. Quando convertemos um objeto de tipo fator para a classe caractere, o resultado é o esperado:

```
# create factor
my_char <- factor(c('a', 'b', 'c'))

# convert and print
print(as.character(my_char))
```

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

Porém, quando fazemos o mesmo procedimento para a classe numérica, o que o R retorna é **longe do esperado**:

```
# set factor
my_values <- factor(5:10)

# convert to numeric (WRONG)
print(as.numeric(my_values))
```

```
R> [1] 1 2 3 4 5 6
```

Esse resultado pode ser explicado pelo fato de que, internamente, fatores são armazenados como índices, indo de 1 até o número total de `Levels`. Essa simplificação minimiza o uso da memória do computador. Quando pedimos ao R para transformar esses fatores em números, ele entende que buscamos o número do índice e não do valor. Para contornar, é fácil: basta transformar o objeto fator em caractere e, depois, em numérico, conforme mostrado a seguir:

```
# converting factors to character and then to numeric
print(as.numeric(as.character(my_values)))
```

```
R> [1] 5 6 7 8 9 10
```



Tenha muito cuidado ao transformar fatores em números. Lembre-se sempre de que o retorno da conversão direta serão os índices dos `levels` e não os valores em si. Esse é um *bug* bem particular que pode ser difícil de identificar em um código complexo.

### 7.3.4 Criando Tabelas de Contingência

Após a criação de um fator, podemos calcular a ocorrência de cada fator com a função `table`. Essa também é chamada de tabela de contingência. Em um caso simples, com apenas um fator, a função `table` conta o número de ocorrências de cada categoria, como a seguir:

```
# create factor
my_factor <- factor(sample(c('Pref', 'Ord'),
```

```

                                size = 20,
                                replace = TRUE))

# print contingency table
print(table(my_factor))

```

```

R> my_factor
R> Ord Pref
R>  9  11

```

Um caso mais avançado do uso de `table` é utilizar mais de um fator para a criação da tabela. Veja o exemplo a seguir:

```

# set factors
my_factor_1 <- factor(sample(c('Pref', 'Ord'),
                             size = 20,
                             replace = TRUE))

my_factor_2 <- factor(sample(paste('Grupo', 1:3),
                             size = 20,
                             replace = TRUE))

# print contingency table with two factors
print(table(my_factor_1, my_factor_2))

```

```

R>           my_factor_2
R> my_factor_1 Grupo 1 Grupo 2 Grupo 3
R>      Ord      2      4      3
R>      Pref      3      4      4

```

A tabela criada anteriormente mostra o número de ocorrências para cada combinação de fator. Essa é uma ferramenta descritiva simples, mas bastante informativa para a análise de grupos de dados.

### 7.3.5 Outras Funções

**levels** - Retorna os Levels de um objeto da classe fator.

```

my_factor <- factor(c('A', 'A', 'B', 'C', 'B'))
print(levels(my_factor))

```

```

R> [1] "A" "B" "C"

```

**as.factor** - Transforma um objeto para a classe fator.

```
my_y <- c('a','b', 'c', 'c', 'a')
my_factor <- as.factor(my_y)
print(my_factor)
```

```
R> [1] a b c c a
R> Levels: a b c
```

**split** - Com base em um objeto de fator, cria uma lista com valores de outro objeto. Esse comando é útil para separar dados de grupos diferentes e aplicar alguma função com `sapply` ou `lapply`.

```
my_factor <- factor(c('A','B','C','C','C','B'))
my_x <- 1:length(my_factor)

my_l <- split(x = my_x, f = my_factor)

print(my_l)
```

```
R> $A
R> [1] 1
R>
R> $B
R> [1] 2 6
R>
R> $C
R> [1] 3 4 5
```

## 7.4 Valores Lógicos

Testes lógicos em dados são centrais no uso do R. Em uma única linha de código podemos testar condições para uma grande quantidade de casos. Esse cálculo é muito utilizado para encontrar casos extremos nos dados (*outliers*) e também para separar diferentes amostras de acordo com algum critério.

### 7.4.1 Criando Valores Lógicos

Em uma sequência de 1 até 10, podemos verificar quais são os elementos maiores que 5 com o seguinte código:

```
# set numerical
my_x <- 1:10

# print a logical test
print(my_x > 5)
```

```
R> [1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
R> [10] TRUE
```

```
# print position of elements from logical test
print(which(my_x > 5))
```

```
R> [1] 6 7 8 9 10
```

A função `which` do exemplo anterior retorna os índices onde a condição é verdadeira (TRUE). O uso do `which` é recomendado quando se quer saber a posição de elementos que satisfazem alguma condição.

Para realizar testes de igualdade, basta utilizar o símbolo de igualdade duas vezes (`==`).

```
# create char
my_char <- rep(c('abc','bcd'), 5)
```

```
# print its contents
print(my_char)
```

```
R> [1] "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc"
R> [10] "bcd"
```

```
# print logical test
print(my_char == 'abc')
```

```
R> [1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
R> [10] FALSE
```

Para o teste de inigualdades, utilizamos o símbolo `!=`:

```
# print inequality test
print(my_char != 'abc')
```

```
R> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
R> [10] TRUE
```

Destaca-se que também é possível testar condições múltiplas, isto é, a ocorrência simultânea de eventos. Utilizamos o operador `&` para esse propósito. Por exemplo: se quiséssemos verificar quais são os valores de uma sequência de 1 a 10 que são maiores que 4 e menores que 7, escreveríamos:

```
my_x <- 1:10
```

```
# print logical for values higher than 4 and lower than 7
print((my_x > 4)&(my_x < 7) )
```

```
R> [1] FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE
R> [10] FALSE
```

```
# print the actual values
idx <- which( (my_x > 4)&(my_x < 7) )
print(my_x[idx])
```

```
R> [1] 5 6
```

Para testar condições não simultâneas, isto é, ocorrências de um ou outro evento, utilizamos o operador `|`. Por exemplo: considerando a sequência anterior, acharíamos os valores maiores que 7 **ou** menores que 4 escrevendo:

```
# location of elements higher than 7 or lower than 4
idx <- which( (my_x > 7)|(my_x < 4) )

# print elements from previous condition
print(my_x[idx])
```

```
R> [1] 1 2 3 8 9 10
```

Observe que, em ambos os casos de uso de testes lógicos, utilizamos parênteses para encapsular as condições lógicas. Poderíamos ter escrito `idx <- which( my_x > 7|my_x < 4 )`, porém o uso do parênteses deixa o código mais claro ao isolar os testes de condições e sinalizar que o resultado da operação será um vetor lógico. Em alguns casos, porém, o uso do parênteses indica hierarquia na ordem das operações e portanto não pode ser ignorado.

Outro uso interessante de objetos lógicos é o teste para saber se um item ou mais pertence a um vetor ou não. Para isso utilizamos o operador `%in%`. Por exemplo, imagine que tens os *tickers* de duas ações, `c('ABC', 'DEF')` e queres saber se é possível encontrar esses tickers na coluna de outra base de dados. Essa é uma operação semelhante ao uso do teste de igualdade, porém em notação vetorial. Veja um exemplo a seguir:

```
library(dplyr)
# location of elements higher than 7 or lower than 4
my_tickers <- c('ABC', 'DEF')

# set df
n_obs <- 100
df_temp <- tibble(tickers = sample(c('ABC', 'DEF', 'GHI', 'JKL'),
                                size = n_obs,
                                replace = TRUE),
                 ret = rnorm(n_obs, sd = 0.05) )
```

```
# find rows with selected tickers
idx <- df_temp$tickers %in% my_tickers

# print elements from previous condition
glimpse(df_temp[idx, ])
```

```
R> Rows: 43
R> Columns: 2
R> $ tickers <chr> "ABC", "ABC", "ABC", "DEF", "DEF", "ABC", ~
R> $ ret <dbl> 0.042864781, 0.017056405, 0.011198439, 0.0~
```

O dataframe mostrado na tela possui dados apenas para ações em `my_tickers`.

## 7.5 Datas e Tempo

A representação e manipulação de datas é um importante aspecto das pesquisas em Finanças e Economia. Manipular datas e horários de forma correta, levando em conta mudanças decorridas de horário de verão, feriados locais, em diferentes zonas de tempo, não é uma tarefa fácil! Felizmente, o R fornece um grande suporte para qualquer tipo de operação com datas e tempo.

Nesta seção estudaremos as funções e classes nativas que representam e manipulam o tempo em R. Aqui, daremos prioridade as funções do pacote `lubridate` (Spinu et al., 2022). Existem, no entanto, muitos pacotes que podem ajudar o usuário a processar objetos do tipo `data` e tempo de forma mais avançada. Caso alguma operação com `data` e tempo não for encontrada aqui, sugiro o estudo dos pacotes `chron` (James and Hornik, 2022), `timeDate` (Wuertz et al., 2022) e `bizdays` (Freitas, 2022).

Antes de começarmos, vale relembrar que toda `data` no R segue o formato ISO 8601 (YYYY-MM-DD), onde YYYY é o ano em quatro números, MM é o mês e DD é o dia. Por exemplo, uma `data` em ISO 8601 é 2022-11-23. Deves familiarizar-se com esse formato pois toda importação de dados com formato de datas diferente desta notação exigirá conversão. Felizmente, essa operação é bastante simples de executar com o `lubridate`.

### 7.5.1 Criando Datas Simples

No R, existem diversas classes que podem representar datas. A escolha entre uma classe de datas e outra baseia-se na necessidade da pesquisa. Em muitas situações não é necessário saber o horário, enquanto que em outras isso é extremamente pertinente pois os dados são coletados ao longo de um dia.

A classe mais básica de datas é `Date`. Essa indica dia, mês e ano, apenas. No

`lubridate`, criamos datas verificando o formato da data de entrada e as funções `ymd` (year-month-date), `dmy` (day-month-year) e `mdy` (month-day-year). Veja a seguir:

```
library(lubridate)

# set Date object
print(ymd('2021-06-24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(dmy('24-06-2021'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(mdy('06-24-2021'))
```

```
R> [1] "2021-06-24"
```

Note que as funções retornam exatamente o mesmo objeto. A diferença no uso é somente pela forma que a data de entrada está estruturada com a posição do dia, mês e ano.

Um benefício no uso das funções do pacote `lubridate` é que as mesmas são inteligentes ao lidar com formatos diferentes. Observe no caso anterior que definimos os elementos das datas com o uso do traço (-) como separador e valores numéricos. Outros formatos também são automaticamente reconhecidos:

```
# set Date object
print(ymd('2021/06/24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(ymd('2021&06&24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(ymd('2021 june 24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(dmy('24 of june 2021'))
```

```
R> [1] "2021-06-24"
```

Isso é bastante útil pois o formato de datas no Brasil é **dia/mês/ano** (DD/MM/YYYY). Ao usar `dmy` para uma data brasileira, a conversão é correta:

```
# set Date from dd/mm/yyyy
my_date <- dmy('24/06/2021')

# print result
print(my_date)
```

```
R> [1] "2021-06-24"
```

Já no pacote `base`, a função correspondente é `as.Date`. O formato da data, porém, deve ser explicitamente definido com argumento `format`, conforme mostrado a seguir:

```
# set Date from dd/mm/yyyy with the definition of format
my_date <- as.Date('24/06/2021', format = '%d/%m/%Y')

# print result
print(my_date)
```

```
R> [1] "2021-06-24"
```

Os símbolos utilizados na entrada `format`, tal como `%d` e `%Y`, são indicadores de formato, os quais definem a forma em que a data a ser convertida está estruturada. Nesse caso, os símbolos `%Y`, `%m` e `%d` definem ano, mês e dia, respectivamente. Existem diversos outros símbolos que podem ser utilizados para processar datas em formatos específicos. Um panorama das principais codificações é apresentado a seguir:

Código	Valor	Exemplo
<code>%d</code>	dia do mês (decimal)	0
<code>%m</code>	mês (decimal)	12
<code>%b</code>	mês (abreviado)	Abr
<code>%B</code>	mês (nome completo)	Abril
<code>%y</code>	ano (2 dígitos)	16
<code>%Y</code>	ano (4 dígitos)	2021

Os símbolos anteriores permitem a criação de datas a partir de variados formatos. Observe como a utilização das funções do `lubridate`, em relação a `base`, são mais simples e fáceis de utilizar, justificando a nossa escolha.

### 7.5.2 Criando Sequências de Datas

Um aspecto interessante no uso de objetos do tipo `Date` é que eles interagem com operações de adição de valores numéricos e com testes lógicos de comparação de datas. Por exemplo: caso quiséssemos adicionar um dia à data `my_date` criada anteriormente, bastaria somar o valor 1 ao objeto:

```
# create date
my_date <- ymd('2021-06-24')

# find next day
my_date_2 <- my_date + 1

# print result
print(my_date_2)
```

```
R> [1] "2021-06-25"
```

A propriedade também funciona com vetores, o que deixa a criação de sequências de datas muito fácil. Nesse caso, o próprio R encarrega-se de verificar o número de dias em cada mês.

```
# create a sequence of Dates
my_date_vec <- my_date + 0:15

# print it
print(my_date_vec)
```

```
R> [1] "2021-06-24" "2021-06-25" "2021-06-26" "2021-06-27"
R> [5] "2021-06-28" "2021-06-29" "2021-06-30" "2021-07-01"
R> [9] "2021-07-02" "2021-07-03" "2021-07-04" "2021-07-05"
R> [13] "2021-07-06" "2021-07-07" "2021-07-08" "2021-07-09"
```

Uma maneira mais customizável de criar sequências de datas é utilizar a função `seq`. Com ela, é possível definir intervalos diferentes de tempo e até mesmo o tamanho do vetor de saída. Caso quiséssemos uma sequência de datas de dois em dois dias, poderíamos utilizar o seguinte código:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-03-20')

# set sequence
my_date_date <- seq(from = my_date_1,
                    to = my_date_2,
                    by = '2 days')
```

```
# print result
print(my_date_date)
```

```
R> [1] "2021-03-07" "2021-03-09" "2021-03-11" "2021-03-13"
R> [5] "2021-03-15" "2021-03-17" "2021-03-19"
```

Caso quiséssemos de duas em duas semanas, escreveríamos:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-04-20')

# set sequence
my_date_date <- seq(from = my_date_1,
                    to = my_date_2,
                    by = '2 weeks')

# print result
print(my_date_date)
```

```
R> [1] "2021-03-07" "2021-03-21" "2021-04-04" "2021-04-18"
```

Outra forma de utilizar `seq` é definir o tamanho desejado do objeto de saída. Por exemplo, caso quiséssemos um vetor de datas com 10 elementos, usaríamos:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-10-20')

# set sequence
my_date_vec <- seq(from = my_date_1,
                  to = my_date_2,
                  length.out = 10)

# print result
print(my_date_vec)
```

```
R> [1] "2021-03-07" "2021-04-01" "2021-04-26" "2021-05-21"
R> [5] "2021-06-15" "2021-07-11" "2021-08-05" "2021-08-30"
R> [9] "2021-09-24" "2021-10-20"
```

O intervalo entre as datas em `my_date_vec` é definido automaticamente pelo R.

### 7.5.3 Operações com Datas

É possível descobrir a diferença de dias entre datas simplesmente diminuindo uma data da outra:

```
# set dates
my_date_1 <- ymd('2015-06-24')
my_date_2 <- ymd('2016-06-24')

# calculate difference
diff_date <- my_date_2 - my_date_1

# print result
print(diff_date)
```

```
R> Time difference of 366 days
```

A saída da operação de subtração é um objeto da classe `diffdate`, o qual possui a classe de lista como sua estrutura básica. Destaca-se que a notação de acesso aos elementos da classe `diffdate` é a mesma utilizada para listas. O valor numérico do número de dias está contido no primeiro elemento de `diff_date`:

```
# print difference of days as numerical value
print(diff_date[[1]])
```

```
R> [1] 366
```

Podemos testar se uma data é maior do que outra com o uso das operações de comparação:

```
# set date and vector
my_date_1 <- ymd('2016-06-20')
my_date_vec <- ymd('2016-06-20') + seq(-5,5)

# test which elements of my_date_vec are older than my_date_1
my_test <- (my_date_vec > my_date_1)

# print result
print(my_test)
```

```
R> [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
R> [10] TRUE TRUE
```

A operação anterior é bastante útil quando se está buscando filtrar um determinado período de tempo nos dados. Nesse caso, basta buscar nas datas o período específico em que estamos interessados e utilizar o objeto lógico da comparação para selecionar os elementos. Veja o exemplo a seguir:

```

library(dplyr)
library(lubridate)

# set first and last dates
first_date <- ymd('2016-06-01')
last_date <- ymd('2016-06-15')

# create `dataframe` and glimpse it
my_temp_df <- tibble(date_vec = ymd('2016-05-25') + seq(0,30),
                    prices=seq(1,10,
                               length.out = length(date_vec)))

glimpse(my_temp_df)

```

```

R> Rows: 31
R> Columns: 2
R> $ date_vec <date> 2016-05-25, 2016-05-26, 2016-05-27, 2016-
R> $ prices <dbl> 1.0, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1, 3~

```

```

# find dates that are between the first and last date
my_idx <- (my_temp_df$date_vec >= first_date) &
  (my_temp_df$date_vec <= last_date)

# use index to filter `dataframe`
my_temp_df_filtered <- my_temp_df %>%
  filter(my_idx) %>%
  glimpse()

```

```

R> Rows: 15
R> Columns: 2
R> $ date_vec <date> 2016-06-01, 2016-06-02, 2016-06-03, 2016-
R> $ prices <dbl> 3.1, 3.4, 3.7, 4.0, 4.3, 4.6, 4.9, 5.2, 5~

```

Nesse caso, o vetor final de preços da coluna `prices` contém apenas informações para o período entre `first_date` e `last_date`.

#### 7.5.4 Lidando com Data e Tempo

O uso da classe `Date` é suficiente quando se está lidando apenas com datas. Em casos em que é necessário levar em consideração o horário, temos que utilizar um objeto do tipo `datetime`.

No pacote `base`, uma das classes utilizadas para esse fim é a `POSIXlt`, a qual armazena o conteúdo de uma data na forma de uma lista. Outra classe que também

é possível utilizar é a `POSIXct`, que armazena as datas como segundos contados a partir de 1970-01-01. Devido ao seu formato de armazenamento, a classe `POSIXct` ocupa menos memória do computador. Junto ao `lubridate`, a classe utilizada para representar data-tempo é `POSIXct` e portanto daremos prioridade a essa. Vale destacar que todos os exemplos apresentados aqui também podem ser replicados para objetos do tipo `POSIXlt`.

O formato tempo/data também segue a norma ISO 8601, sendo representado como `ano-mês-dia horas:minutos:segundos zonadetempo` (YYYY-MM-DD HH:mm:ss TMZ). Veja o exemplo a seguir:

```
# creating a POSIXct object
my_timedate <- as.POSIXct('2021-01-01 16:00:00')

# print result
print(my_timedate)
```

```
R> [1] "2021-01-01 16:00:00 -03"
```

O pacote `lubridate` também oferece funções inteligentes para a criação de objetos do tipo data-tempo. Essas seguem a mesma linha de raciocínio que as funções de criar datas. Veja a seguir:

```
library(lubridate)

# creating a POSIXlt object
my_timedate <- ymd_hms('2021-01-01 16:00:00')

# print it
print(my_timedate)
```

```
R> [1] "2021-01-01 16:00:00 UTC"
```

Destaca-se que essa classe adiciona automaticamente o fuso horário. Caso seja necessário representar um fuso diferente, é possível fazê-lo com o argumento `tz`:

```
# creating a POSIXlt object with custom timezone
my_timedate_tz <- ymd_hms('2021-01-01 16:00:00',
                          tz = 'GMT')

# print it
print(my_timedate_tz)
```

```
R> [1] "2021-01-01 16:00:00 GMT"
```

É importante ressaltar que, para o caso de objetos do tipo `POSIXlt` e `POSIXct`, **as operações de soma e diminuição referem-se a segundos** e não dias, como no

caso do objeto da classe `Date`.

```
# Adding values (seconds) to a POSIXlt object and printing it
print(my_timedate_tz + 30)
```

```
R> [1] "2021-01-01 16:00:30 GMT"
```

Assim como para a classe `Date`, existem símbolos específicos para lidar com componentes de um objeto do tipo data/tempo. Isso permite a formatação customizada de datas. A seguir, apresentamos um quadro com os principais símbolos e os seus respectivos significados.

Código	Valor	Exemplo
%H	Hora (decimal, 24 horas)	23
%I	Hora (decimal, 12 horas)	11
%M	Minuto (decimal, 0-59)	12
%p	Indicador AM/PM	AM
%S	Segundos (decimal, 0-59)	50

A seguir veremos como utilizar essa tabela para customizar datas.

### 7.5.5 Personalizando o Formato de Datas

A notação básica para representar datas e data/tempo no R pode não ser a ideal em algumas situações. No Brasil, por exemplo, indicar datas no formato `YYYY-MM-DD` pode gerar bastante confusão em um relatório formal. É recomendado, portanto, modificar a representação das datas para o formato esperado, isto é, `DD/MM/YYYY`.

Para formatar uma data, utilizamos a função `format`. Seu uso baseia-se nos símbolos de data e de horário apresentados anteriormente. A partir desses, pode-se criar qualquer customização. Veja o exemplo a seguir, onde apresenta-se a modificação de um vetor de datas para o formato brasileiro:

```
# create vector of dates
my_dates <- seq(from = ymd('2021-01-01'),
               to = ymd('2021-01-15'),
               by = '1 day')

# change format
my_dates_br <- format(my_dates, '%d/%m/%Y')

# print result
print(my_dates_br)
```

```
R> [1] "01/01/2021" "02/01/2021" "03/01/2021" "04/01/2021"
R> [5] "05/01/2021" "06/01/2021" "07/01/2021" "08/01/2021"
R> [9] "09/01/2021" "10/01/2021" "11/01/2021" "12/01/2021"
R> [13] "13/01/2021" "14/01/2021" "15/01/2021"
```

O mesmo procedimento pode ser realizado para objetos do tipo data/tempo (POSIXct):

```
# create vector of date-time
my_datetime <- ymd_hms('2021-01-01 12:00:00') + seq(0,560,60)

# change to Brazilian format
my_dates_br <- format(my_datetime, '%d/%m/%Y %H:%M:%S')

# print result
print(my_dates_br)
```

```
R> [1] "01/01/2021 12:00:00" "01/01/2021 12:01:00"
R> [3] "01/01/2021 12:02:00" "01/01/2021 12:03:00"
R> [5] "01/01/2021 12:04:00" "01/01/2021 12:05:00"
R> [7] "01/01/2021 12:06:00" "01/01/2021 12:07:00"
R> [9] "01/01/2021 12:08:00" "01/01/2021 12:09:00"
```

Pode-se também customizar para formatos bem específicos. Veja a seguir:

```
# set custom format
my_dates_custom <- format(my_dates,
                           'Year=%Y | Month=%m | Day=%d')

# print result
print(my_dates_custom)
```

```
R> [1] "Year=2021 | Month=01 | Day=01"
R> [2] "Year=2021 | Month=01 | Day=02"
R> [3] "Year=2021 | Month=01 | Day=03"
R> [4] "Year=2021 | Month=01 | Day=04"
R> [5] "Year=2021 | Month=01 | Day=05"
R> [6] "Year=2021 | Month=01 | Day=06"
R> [7] "Year=2021 | Month=01 | Day=07"
R> [8] "Year=2021 | Month=01 | Day=08"
R> [9] "Year=2021 | Month=01 | Day=09"
R> [10] "Year=2021 | Month=01 | Day=10"
R> [11] "Year=2021 | Month=01 | Day=11"
R> [12] "Year=2021 | Month=01 | Day=12"
R> [13] "Year=2021 | Month=01 | Day=13"
```

```
R> [14] "Year=2021 | Month=01 | Day=14"
```

```
R> [15] "Year=2021 | Month=01 | Day=15"
```

### 7.5.6 Extrair Elementos de uma Data

Para extrair elementos de datas tal como o ano, mês, dia, hora, minuto e segundo, uma alternativa é utilizar função `format`. Observe o próximo exemplo, onde recuperamos apenas as horas de um objeto `POSIXct`:

```
library(lubridate)

# create vector of date-time
my_datetime <- seq(from = ymd_hms('2021-01-01 12:00:00'),
                  to = ymd_hms('2021-01-01 18:00:00'),
                  by = '1 hour')

# get hours from POSIXlt
my_hours <- as.numeric(format(my_datetime, '%H'))

# print result
print(my_hours)
```

```
R> [1] 12 13 14 15 16 17 18
```

Da mesma forma, poderíamos utilizar os símbolos `%M` e `%S` para recuperar facilmente minutos e segundos de um vetor de objetos `POSIXct`.

```
# create vector of date-time
my_datetime <- seq(from = ymd_hms('2021-01-01 12:00:00'),
                  to = ymd_hms('2021-01-01 18:00:00'),
                  by = '15 min')

# get minutes from POSIXlt
my_minutes <- as.numeric(format(my_datetime, '%M'))

# print result
print(my_minutes)
```

```
R> [1] 0 15 30 45 0 15 30 45 0 15 30 45 0 15 30 45 0 15
```

```
R> [19] 30 45 0 15 30 45 0
```

Outra forma é utilizar as funções do `lubridate`, tal como `hour` e `minute`:

```
# get hours with lubridate
print(hour(my_datetime))
```

```
R> [1] 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15 16 16
R> [19] 16 16 17 17 17 17 18
```

```
# get minutes with lubridate
print(minute(my_datetime))
```

```
R> [1] 0 15 30 45 0 15 30 45 0 15 30 45 0 15 30 45 0 15
R> [19] 30 45 0 15 30 45 0
```

Outras funções também estão disponíveis para os demais elementos de um objeto data-hora.

### 7.5.7 Conhecendo o Horário e a Data Atual

O R inclui várias funções que permitem o usuário utilizar no seu código o horário e data atual do sistema. Isso é bastante útil quando se está criando registros e é importante que a data e horário de execução do código seja conhecida futuramente.

Para conhecer o dia atual, basta utilizarmos a função `base::Sys.Date` ou `lubridate::today`:

```
library(lubridate)

# get today
print(Sys.Date())
```

```
R> [1] "2022-11-23"
```

```
# print it
print(today())
```

```
R> [1] "2022-11-23"
```

Para descobrir a data e horário, utilizamos a função `base::Sys.time` ou `lubridate::now`:

```
# get time!
print(Sys.time())
```

```
R> [1] "2022-11-23 10:57:03 -03"
```

```
# get time!
print(now())
```

```
R> [1] "2022-11-23 10:57:03 -03"
```

Com base nessas, podemos escrever:

```
library(stringr)

# example of log message
my_str <- str_c('This code was executed in ', now())

# print it
print(my_str)
```

```
R> [1] "This code was executed in 2022-11-23 10:57:03"
```

### 7.5.8 Outras Funções Úteis

**weekdays** - Retorna o dia da semana de uma ou várias datas.

```
# set date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-01-5'),
                by = '1 day')

# find corresponding weekdays
my_weekdays <- weekdays(my_dates)

# print it
print(my_weekdays)
```

```
R> [1] "Friday" "Saturday" "Sunday" "Monday" "Tuesday"
```

**months** - Retorna o mês de uma ou várias datas.

```
# create date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-12-31'),
                by = '1 month')

# find months
my_months <- months(my_dates)

# print result
print(my_months)
```

```
R> [1] "January" "February" "March" "April"
R> [5] "May" "June" "July" "August"
R> [9] "September" "October" "November" "December"
```

**quarters** - Retorna a localização de uma ou mais datas dentro dos quartis do ano.

```
# get quartiles of the year
my_quarters <- quarters(my_dates)
print(my_quarters)
```

```
R> [1] "Q1" "Q1" "Q1" "Q2" "Q2" "Q2" "Q3" "Q3" "Q3" "Q4" "Q4"
R> [12] "Q4"
```

**OlsonNames** - Retorna um vetor com as zonas de tempo disponíveis no R. No total, são mais de 500 itens. Aqui, apresentamos apenas os primeiros cinco elementos.

```
# get possible timezones
possible_tz <- OlsonNames()

# print it
print(possible_tz[1:5])
```

```
R> [1] "Africa/Abidjan"      "Africa/Accra"
R> [3] "Africa/Addis_Ababa" "Africa/Algiers"
R> [5] "Africa/Asmara"
```

**Sys.timezone** - Retorna a zona de tempo do sistema.

```
# get current timezone
print(Sys.timezone())
```

```
R> [1] "America/Sao_Paulo"
```

**cut** - Retorna um fator a partir da categorização de uma classe de data e tempo.

```
# set example date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-03-01'),
                by = '5 days')

# group vector based on monthly breaks
my_month_cut <- cut(x = my_dates,
                   breaks = 'month')

# print result
print(my_month_cut)
```

```
R> [1] 2021-01-01 2021-01-01 2021-01-01 2021-01-01 2021-01-01
R> [6] 2021-01-01 2021-01-01 2021-02-01 2021-02-01 2021-02-01
R> [11] 2021-02-01 2021-02-01
R> Levels: 2021-01-01 2021-02-01
```

```
# set example datetime vector
my_datetime <- as.POSIXlt('2021-01-01 12:00:00') + seq(0,250,15)

# set groups for each 30 seconds
my_cut <- cut(x = my_datetime, breaks = '30 secs')

# print result
print(my_cut)
```

```
R> [1] 2021-01-01 12:00:00 2021-01-01 12:00:00
R> [3] 2021-01-01 12:00:30 2021-01-01 12:00:30
R> [5] 2021-01-01 12:01:00 2021-01-01 12:01:00
R> [7] 2021-01-01 12:01:30 2021-01-01 12:01:30
R> [9] 2021-01-01 12:02:00 2021-01-01 12:02:00
R> [11] 2021-01-01 12:02:30 2021-01-01 12:02:30
R> [13] 2021-01-01 12:03:00 2021-01-01 12:03:00
R> [15] 2021-01-01 12:03:30 2021-01-01 12:03:30
R> [17] 2021-01-01 12:04:00
R> 9 Levels: 2021-01-01 12:00:00 ... 2021-01-01 12:04:00
```

## 7.6 Dados Omissos - NA (*Not available*)

Uma das principais inovações do R em relação a outras linguagens de programação é a representação de dados omissos através de objetos da classe NA (*Not Available*). A falta de dados pode ter inúmeros motivos, tal como a falha na coleta de informações ou simplesmente a não existência dos mesmos. Esses casos são tratados por meio da remoção ou da substituição dos dados omissos antes realizar uma análise mais profunda. A identificação desses casos, portanto, é de extrema importância.

### 7.6.1 Definindo Valores NA

Para definirmos os casos omissos nos dados, basta utilizar o símbolo NA:

```
# a vector with NA
my_x <- c(1, 2, NA, 4, 5)

# print it
print(my_x)
```

```
R> [1] 1 2 NA 4 5
```

Vale destacar que a operação de qualquer valor NA com outro sempre resultará em NA.

```
# example of NA interacting with other objects
print(my_x + 1)
```

```
R> [1] 2 3 NA 5 6
```

Isso exige cuidado quando se está utilizando alguma função com cálculo recursivo, tal como `cumsum` e `cumprod`. Nesses casos, todo valor consecutivo ao NA será transformado em NA. Veja os exemplos a seguir com as duas funções:

```
# set vector with NA
my_x <- c(1:5, NA, 5:10)

# print cumsum (NA after sixth element)
print(cumsum(my_x))
```

```
R> [1] 1 3 6 10 15 NA NA NA NA NA NA NA
```

```
# print cumprod (NA after sixth element)
print(cumprod(my_x))
```

```
R> [1] 1 2 6 24 120 NA NA NA NA NA NA NA
```



Toda vez que utilizar as funções `cumsum` e `cumprod`, certifique-se de que não existe algum valor NA no vetor de entrada. Lembre-se de que todo NA é contagioso e o cálculo recursivo irá resultar em um vetor repleto de dados faltantes.

### 7.6.2 Encontrando e Substituindo Valores NA

Para encontrar os valores NA em um vetor, basta utilizar a função `is.na`:

```
# set vector with NA
my_x <- c(1:2, NA, 4:10)

# find location of NA
idx_na <- is.na(my_x)
print(idx_na)
```

```
R> [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
R> [10] FALSE
```

Para substituí-los, use indexação com a saída de `is.na`:

```
# set vector
my_x <- c(1, NA, 3:4, NA)
```

```
# replace NA for 2
my_x[is.na(my_x)] <- 2

# print result
print(my_x)
```

```
R> [1] 1 2 3 4 2
```

Outra maneira de limpar o objeto é utilizar a função `na.omit`, que retorna o mesmo objeto mas sem os valores NA. Note, porém, que o tamanho do vetor irá mudar e o objeto será da classe `omit`, o que indica que o vetor resultante não inclui os NA e apresenta, também, a posição dos elementos NA encontrados.

```
# set vector
my_char <- c(letters[1:3], NA, letters[5:8])

# print it
print(my_char)
```

```
R> [1] "a" "b" "c" NA "e" "f" "g" "h"
```

```
# use na.omit to remove NA
my_char <- na.omit(my_char)

# print result
print(my_char)
```

```
R> [1] "a" "b" "c" "e" "f" "g" "h"
R> attr("na.action")
R> [1] 4
R> attr("class")
R> [1] "omit"
```

Apesar de o tipo de objeto ter sido trocado, devido ao uso de `na.omit`, as propriedades básicas do vetor inicial se mantêm. Por exemplo: o uso de `nchar` no objeto resultante é possível.

```
# trying nchar on a na.omit object
print(nchar(my_char))
```

```
R> [1] 1 1 1 1 1 1 1
```

Para outros objetos, porém, recomenda-se cautela quando no uso da função `na.omit`.

### 7.6.3 Outras Funções Úteis

**complete.cases** - Retorna um vetor lógico que indica se as linhas do objeto possuem apenas valores não omissos. Essa função é usada exclusivamente para **dataframes** e matrizes.

```
# create matrix
my_mat <- matrix(1:15, nrow = 5)

# set an NA value
my_mat[2,2] <- NA

# print index with rows without NA
print(complete.cases(my_mat))
```

```
R> [1] TRUE FALSE TRUE TRUE TRUE
```

## 7.7 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Considere os seguintes os vetores x e y:

```
set.seed(7)
x <- sample (1:3, size = 5, replace = T)
y <- sample (1:3, size = 5, replace = T)
```

Qual é a soma dos elementos de um novo vetor resultante da multiplicação entre os elementos de x e y?

- a) 41
- b) 34
- c) 55
- d) 48
- e) 31

02 - Caso realizássemos uma soma cumulativa de uma sequência entre 1 e 100, em qual elemento esta soma iria passar de 50?

- a) 5
- b) 3
- c) 1
- d) 7
- e) 10

03 - Utilizando o R, crie uma sequência em objeto chamado **seq\_1** entre -15 e 10, onde o intervalo entre valores é sempre igual a 2. Qual o valor da soma dos elementos

de `seq_1`?

- a) -15
- b) -6
- c) -39
- d) -31
- e) -23

04 - Defina outro objeto chamado `seq_2` contendo uma sequência de tamanho 1000, com valores entre 0 e 100. Qual é o desvio padrão (função `sd`) dessa sequência?

- a) 28.91
- b) 19.58
- c) 12.45
- d) 38.46
- e) 67.37

05 - Calcule a soma entre vetores `seq_1` e `seq_2` (veja exercício anterior). Esta operação funcionou apesar do tamanho diferente dos vetores? Explique sua resposta. Caso funcionar, qual o maior valor do vetor resultante?

- a) 148.2
- b) 169.3
- c) 127.1
- d) 109.0
- e) 190.4

06 - Vamos supor que, em certa data, você comprou 100 ações de uma empresa, a `price_purchase` reais por ação. Depois de algum tempo, você vendeu 30 ações por 18 reais cada e as 70 ações restantes foram vendidas por 22 reais em um dia posterior. Usando um *script* em R, estruture este problema financeiro criando objetos numéricos. Qual é o lucro bruto desta transação no mercado de ações?

- a) R\$922
- b) R\$1.037
- c) R\$692
- d) R\$807
- e) R\$580

07 - Crie um vetor `x` de acordo com a fórmula a seguir, onde  $i = 1 \dots 100$ . Qual é o valor da soma dos elementos de `x`?

$$x_i = \frac{-1^{i+1}}{2i-1}$$

- a) 0.7829
- b) 2.2385

- c) 0.3014
- d) 0.5253
- e) 1.4556

08 - Crie um vetor  $z_i$  de acordo com a fórmula a seguir onde  $x_i = 1...50$  e  $y_i = 50...1$ . Qual é o valor da soma dos elementos de  $z_i$ ? Dica: veja o funcionamento da função `dplyr::lag`.

$$z_i = \frac{y_i - x_{i-1}}{y_{i-2}}$$

- a) -27.78
- b) -65.96
- c) -82.23
- d) -44.80
- e) -148.19

09 - Usando uma semente de valor 11 em `set.seed()`, crie um objeto chamado `x` com valores aleatórios da distribuição Normal com média igual a 10 e desvio padrão igual a 10. Usando função `cut`, crie outro objeto que defina dois grupos com base em valores de `x` maiores que 15 e menores que 15. Qual a quantidade de observações no primeiro grupo?

- a) 504
- b) 92
- c) 688
- d) 229
- e) 367

10 - Crie o seguinte objeto com o código a seguir:

```
set.seed(15)
my_char <- paste(sample(letters, 5000, replace = T),
                 collapse = '')
```

Qual a quantidade de vezes que a letra 'x' é encontrada no objeto de texto `my_char`?

- a) 297
- b) 264
- c) 198
- d) 174
- e) 231

11 - Baseado no objeto `my_char` criado anteriormente, caso dividíssemos o mesmo em diversos pedaços menores utilizando a letra "b", qual é o número de caracteres no **maior** pedaço encontrado?

- a) 188
- b) 125
- c) 167
- d) 146
- e) 110

12 - No endereço <https://www.gutenberg.org/files/1342/1342-0.txt> é possível acessar um arquivo .txt contendo o texto integral do livro *Pride and Prejudice* de Jane Austen. Utilize funções `download.file` e `readr::read_lines` para importar o livro inteiro como um vetor de caracteres chamado `my_book` no R. Quantas linhas o objeto resultante possui?

- a) 5029
- b) 3675
- c) 8704
- d) 2537
- e) 1698

13 - Junte o vetor de caracteres em `my_book` para um único valor (texto) em outro objeto chamado `full_text` usando função `paste0(my_book, collapse = '\n')`. Utilizando este último e pacote `stringr`, quantas vezes a palavra 'King' é repetida na totalidade do texto?

- a) 55
- b) 93
- c) 70
- d) 30
- e) 163

14 - Para o objeto `full_text` criado anteriormente, utilize função `stringr::str_split` para *quebrar* o texto inteiro em função de espaços em branco. Com base nesse, crie uma tabela de frequência. Qual a palavra mais utilizada no texto? Dica: Remova todos os casos de caracteres vazios ('').

- a) and
- b) to
- c) you
- d) my
- e) the

15 - Assumindo que uma pessoa nascida em 2000-05-12 irá viver por 100 anos, qual é o número de dias de aniversário que cairão em um final de semana (sábado ou domingo)? Dica: use operador `%in%` para checar uma condição múltipla nos dados.

- a) 0
- b) 0

- c) 0
- d) 0
- e) 0

16 - Qual data e horário é localizado  $10^4$  **segundos** após 2021-02-02 11:50:02?

- a) 2021-02-02 12:44:52
- b) 2021-02-02 10:23:31
- c) 2021-02-02 10:29:10
- d) 2021-02-02 14:36:42
- e) 2021-02-02 10:48:06

# Capítulo 8

## Programando com o R



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.1 Criando Funções



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.2 Utilizando Loops (comando *for*)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.3 Execuções Condicionais (if, else, switch)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.4 Utilizando as Funções da Família apply



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 8.4.1 Função lapply



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 8.4.2 Função sapply



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.4.3 Função `tapply`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.4.4 Função `mapply`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.4.5 Função `apply`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 8.5 Utilizando Pacote `purrr`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.5.1 Função `purrr::map`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.5.2 Função `purrr::safely`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.5.3 Função `purrr::pmap`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 8.6 Manipulação de Dados com `dplyr`



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.6.1 Operações de Grupo com dplyr



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 8.6.2 Operações de Grupo Complexas com dplyr



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 8.7 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Crie uma função chamada `say_my_name` que tome como entrada um nome de pessoa e mostre na tela o texto *Your name is ...*. Dentro do escopo da função, utilize comentários para descrever o propósito da função, suas entradas e saídas.

02 - Implemente um teste para os objetos de entrada, de forma que, quando o nome de entrada não for da classe `character`, um erro é retornado ao usuário. Teste sua nova função e verifique se a mesma está funcionando conforme esperado.

03 - Crie um vetor com cinco nomes quaisquer, chamado `my_names`. Utilizando um *loop*, aplique função `say_my_name` para cada elemento de `my_names`.

04 - No banco de dados do Brasil.IO<sup>1</sup> encontrará uma tabela com nomes e gêneros derivados de uma das pesquisas do IBGE. Importe os dados do arquivo para R e, usando um *loop*, aplique a função `say_my_name` a 15 nomes aleatórios do banco de dados. Dica: neste caso, você pode baixar os dados direto do link usando `readr::read_csv(LINK)`.

05 - Refaça o exercício anterior utilizando comandos `sapply` ou `purrr::map`.

06 - Use o pacote `BatchGetSymbols` para baixar dados do índice SP500 (`'^GSPC'`), Ibovespa (`'^BVSP'`), FTSE (`'^FSTE'`) e Nikkei 225 (`'^N225'`) de '2010-01-01' até a data atual. Com os dados importados, use um *loop* para calcular o retorno

<sup>1</sup><https://data.brasil.io/dataset/genero-nomes/grupos.csv.gz>

médio, máximo e mínimo de cada índice durante o período analisado. Salve todos os resultados em uma tabela única e a mostre no *prompt* do R.

07 - Refaça o exercício anterior utilizando as funções `group_by` e `summarise`, ambas do pacote `dplyr`.

08 - No site do Rstudio CRAN logs<sup>2</sup> você encontrará dados sobre as estatísticas de download para a distribuição de base de R na seção *Daily R downloads*. Usando suas habilidades de programação, importe todos os dados disponíveis entre 2020-01-01 e 2020-01-15 e agregue-os em uma única tabela. Qual país apresenta a maior contagem de downloads do R?

- a) HU
- b) JM
- c) MD
- d) CR
- e) US

---

<sup>2</sup><http://cran-logs.rstudio.com/>

# Capítulo 9

## Estruturando e Limpando Dados



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 9.1 O Formato do dataframe



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 9.1.1 Conversão entre *long* e *wide*



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 9.2 Convertendo Listas em dataframes



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 9.3 Eliminando Outliers



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 9.4 Desinflationando Dados de Preços



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 9.5 Modificando a Frequência Temporal dos Dados



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 9.6 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Considere o `dataframe` criado com o código a seguir:

```
library(tidyverse)

my_N <- 100

df <- bind_rows(tibble(ticker = rep('STOCK 1', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))),
               tibble(ticker = rep('STOCK 2', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))) )

print(df)
```

O formato do `dataframe` é longo ou largo? Explique sua resposta.

02 - Modifique o formato do `dataframe` anterior, de longo para largo ou vice-versa.

```
library(tidyverse)

my_N <- 100

df <- bind_rows(tibble(ticker = rep('STOCK 1', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))),
               tibble(ticker = rep('STOCK 2', my_N),
                        ref_date = Sys.Date() + 1:my_N,
                        price = 100 + cumsum(rnorm(my_N))) )

print(df)
```

03 - Considere a seguinte lista:

```
library(tidyverse)

my_l <- list(df1 = tibble(x = 1:100, y = runif(100)),
            df2 = tibble(x = 1:100, y = runif(100), v = runif(100)),
            df3 = tibble(x = 1:100, y = runif(100), z = runif(100)) )
```

Agrege todos `dataframes` em `my_l` para um objeto único usando funções `do.call` ou `dplyr::bind_rows`. O que aconteceu com os dados de `df1` onde colunas `v` e `z` não existem?

04 - Utilize pacote `BatchGetSymbols` para baixar os dados do índice SP500 (`'^GSPC'`) desde 1950-01-01 até 2021-01-01. Quais é a soma dos 5 maiores retornos positivos do índice?

- a) 0.5834
- b) 0.7691
- c) 0.5014
- d) 0.6763
- e) 0.8620

05 - Use função `replace_outliers` (veja seção 9.3), criada neste capítulo, para remover *outliers* de todas as colunas numéricas dos dados do SP500 importados anteriormente com `my_prob = 0.025`. Quantas linhas foram perdidas neste processo de limpeza?

- a) 5574
- b) 4101
- c) 7281
- d) 2627
- e) 1154

06 - Use a função `BatchGetSymbols::BatchGetSymbols` para baixar os preços do índice FTSE ('^FTSE') de 2010-01-01 até 2021-01-01. Em seguida, reconstrua os dados na frequência anual, definindo cada valor do ano como sendo a última observação do período. Dica: veja a função `dplyr::summary_all` para uma forma funcional de agregar todas as colunas de um `dataframe`.

07 - Use os mesmos dados diários do FTSE e reconstrua os dados na frequência mensal, novamente utilizando a última observação do período.

08 - Para os mesmos dados diários do FTSE, verifique as datas e preços das 20 maiores quedas de preços. Se, para cada um desses casos, um investidor comprasse o índice no preço das maiores quedas e o mantivesse por 30 dias, qual seria seu retorno nominal médio por transação?

- a) 2,53%
- b) 3,85%
- c) 2,82%
- d) 3,34%
- e) 4,37%

# Capítulo 10

## Visualizando Dados



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.1 Criando Janelas de Gráficos com x11



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.2 Criando Figuras com qplot



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 10.3 Criando Figuras com ggplot



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.3.1 A Curva de Juros Brasileira



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.3.2 Usando Temas



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.3.3 Criando Painéis com facet\_wrap



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 10.4 Uso do Operador *pipeline*



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 10.5 Criando Figuras Estatísticas



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.5.1 Criando Histogramas



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.5.2 Criando Figuras *Boxplot*



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 10.5.3 Criando Figuras QQ



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 10.6 Salvando Figuras para Arquivos



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 10.7 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Baixe dados da ação CIEL3 com o pacote `BatchGetSymbols` para os últimos 500 dias. Apresente o gráfico de linha do preço ajustado ao longo do tempo utilizando função `ggplot2::ggplot`. Tenha certeza que:

- Eixos x e y estão corretamente nomeados
- O gráfico tem um título (“Preços para CIEL3”), subtítulo (“Dados de YYYY-MM-DD até YYYY-MM-DD”) e um *caption* (“Solução para exercício 01, cap 10 - adfeR”).

02 - Baixe dados das ações PETR3 (PETR3.SA), VALE3 (VALE3.SA), GGBR4 (GGBR4.SA) com `BatchGetSymbols` para os últimos 1500 dias. Apresente, no mesmo gráfico, os preços das ações com diferentes cores de linhas. Mantenha todos demais aspectos do gráfico anterior.

03 - Para o gráfico anterior, adicione pontos nas linhas.

04 - Para o mesmo gráfico, separe os preços das ações em painéis diferentes com a função `ggplot2::facet_wrap`. Use argumento `scales = 'free'` para “soltar” os eixos de cada figura.

05 - Modifique o tema do gráfico anterior para uma escala em preto e branco, tanto para a área do gráfico quanto para as cores das linhas.

06 - Para os dados anteriores, apresente o histograma dos retornos das diferentes ações em painéis diferentes e salve o resultado em um arquivo chamado 'histograms.png'.

07 - Use a função `BatchGetSymbols::GetIbovStocks` para descobrir todos os tickers pertencentes ao índice Ibovespa atualmente. Usando `BatchGetSymbols`, baixe os dados de retorno anual para todas as ações do índice de 2015 até o dia atual. Depois disso, crie o mapa de média/variância plotando o retorno anual médio em relação ao seu desvio padrão. Dica: Use a opção paralela em `BatchGetSymbols` para acelerar a execução. Você encontrará muitos retornos discrepantes nos dados brutos. Certifique-se de que o gráfico esteja visível limitando os eixos x e y (consulte as funções `ggplot2::xlim` e `ggplot2::ylim`).



# Capítulo 11

## Econometria Financeira com o R



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.1 Modelos Lineares (OLS)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 11.1.1 Simulando um Modelo Linear



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.1.2 Estimando um Modelo Linear



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.1.3 Inferência Estatística em Modelos Lineares



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.2 Modelos Lineares Generalizados (GLM)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.2.1 Simulando um Modelo GLM



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.2.2 Estimando um Modelo GLM



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.3 Modelos para Dados em Painel



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.3.1 Simulando Dados em Painel



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.3.2 Estimando Modelos de Dados em Painel



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.4 Modelos ARIMA



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.4.1 Simulando Modelos ARIMA



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.4.2 Estimando Modelos ARIMA



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.4.3 Prevendo Modelos ARIMA



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.5 Modelos GARCH



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.5.1 Simulando Modelos GARCH



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.5.2 Estimando Modelos GARCH



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.5.3 Prevendo Modelos GARCH



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.6 Modelos de Mudança de Regime



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.6.1 Simulando Modelos de Mudança de Regime



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.6.2 Estimando Modelos de Mudança de Regime



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 11.6.3 Prevendo Modelos de Mudança de Regime



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.7 Trabalhando com Diversos Modelos



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 11.8 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Simule o seguinte processo linear no R:

```
set.seed(5)

# number of obs
n_row <- 100

# set x as Normal (0, 1)
x <- rnorm(n_row)

# set coefficients
my_alpha <- 1.5
my_beta <- 0.5

# build y
y <- my_alpha + my_beta*x + rnorm(n_row)
```

A partir de  $x$  e  $y$ , estime um modelo linear onde  $x$  é a variável explicativa e  $y$  é a variável explicada. Use função `summary` no objeto de retorno da estimação para obter mais detalhes sobre o modelo. Qual é o valor do beta estimado dos dados simulados?

- a) 0.4003
- b) 1.5038
- c) 0.8707
- d) 0.2910
- e) 0.6331

02 - Utilizando pacote `car`, teste a hipótese conjunta de que o valor de alpha é igual a 1.5 e beta igual a 0.5. Qual o valor do teste F resultante?

- a) 10.727

- b) 16.967
- c) 7.799
- d) 40.301
- e) 23.335

03 - Utilize pacote `gvlma` para testar as premissas do OLS para o modelo estimado anteriormente. O modelo passa em todos os testes? Em caso negativo, aumente o valor de `n_row` para 1000 e tente novamente. O aumento do número de observações do modelo impactou no teste das premissas? De que forma?

04 - Utilize função `BatchGetSymbols::GetSP500Stocks` para baixar dados de todas ações pertencentes ao atual índice SP500 para os últimos três anos. Usando o SP500 como o índice de mercado, calcule o beta para cada uma das ações. Apresente o histograma dos *betas* estimados. Note que os retornos do SP500 (`'^GSPC'`) não estão disponíveis na base de dados original e devem ser baixados e agregados a base de dados original.

05 - Para os dados importados anteriormente, estime uma versão em dados de painel para o modelo de mercado (beta). Nesta versão, cada ação possui um intercepto diferente, porém compartilham o mesmo beta. O beta estimado é significativo a 5%?

06 - Utilizando as funções do tidyverse, `dplyr::group_by` e `dplyr::do`, estime um modelo ARIMA para os retornos de cada ação dos dados importados no exercício do SP500. No mesmo `dataframe` de saída, crie uma nova coluna com a previsão em  $t+1$  de cada modelo. Qual ação possui maior expectativa de retorno para  $t+1$ ?

07 - No mesmo código utilizado na questão anterior, adicione uma nova coluna-lista com a estimação de um modelo ARMA(1,0)-GARCH(1,1) para os retornos de cada ação. Adicione outra coluna com a previsão de volatilidade (desvio padrão) em  $t+1$ . Ao dividir o retorno esperado calculado no item anterior pelo risco previsto, temos um índice de direção do mercado, onde aquelas ações com maior valor de índice apresentam maior retorno esperado por menor risco. Qual ação é mais atrativa e possui maior valor deste índice?

08 - Para a mesma base de dados do SP500, selecione 4 ações aleatoriamente e estime um modelo de mudança de regime markoviano equivalente ao apresentado no item 11.6 para cada ação. Utilize função `plot` para apresentar o gráfico das probabilidades suavizadas e salve cada figura em uma pasta chamada `'fig'`.

# Capítulo 12

## Reportando Resultados e Criando Relatórios



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 12.1 Reportando Tabelas



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 12.2 Reportando Modelos



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 12.3 Criando Relatórios com o *RMarkdown*



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 12.4 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Observe os dados disponíveis no arquivo `grunfeld.csv`. Importe os dados no R e monte uma tabela descritiva das variáveis. Essa tabela deve oferecer informações suficientes para o leitor entender os dados. Utilize pacote `xtable` para reportar a mesma no formato *LaTeX* ou Word/Writer.

02 - Utilizando função `BatchGetSymbols::GetSP500Stocks`, selecione 4 ações aleatoriamente e baixe os dados de preços ajustados para os últimos três anos. Estime um modelo ARIMA(1, 0, 1) para cada ação e reporte o resultado na tela do R com função `texreg::screenreg`.

03 - Crie um novo relatório em *Rmarkdown* contemplando os dois exercícios anteriores. Compile o relatório em html e abra o mesmo no seu *browser*.

# Capítulo 13

## Otimização de Código



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.1 Otimizando Código em R



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.2 Otimizando a Velocidade de Execução



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.2.1 Perfil do Código R (*profiling*)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.2.2 Estratégias para Melhorar a Velocidade de Execução



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 13.2.2.1 Use Operações Vetoriais



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 13.2.2.2 Junção Repetitiva de dataframes



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.2.3 Usando Código C ++ (pacote Rcpp)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

### 13.2.4 Usando Cacheamento Local (pacote memoise)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

#### 13.2.4.1 Usando Processamento Paralelo (pacote furr)



Você chegou ao fim da versão online do livro **Análise de Dados Financeiros e Econômicos com o R**, terceira edição. O conteúdo integral da obra pode ser adquirido na loja da Amazon como ebook ou livro impresso. A compra do livro é uma ótima maneira de suportar este e outros projetos do autor.

## 13.3 Exercícios

Todas soluções de exercícios estão disponíveis em <https://www.msperlin.com/adfeR>.

01 - Considere o seguinte código:

```
library(tidyverse)
library(forecast)
library(BatchGetSymbols)

ticker <- '^GSPC'
df_prices <- BatchGetSymbols(tickers = ticker,
                             first.date = '2010-01-01')[[2]]

my_arima <- auto.arima(df_prices$ret.adjusted.prices)
summary(my_arima)
```

Use funções `Rprof` e `profvis` para identificar o gargalo do código. Qual número da linha que está tomando maior tempo de execução?

02 - Use o pacote `Rcpp` para escrever e usar uma função em linguagem C++ que irá adicionar elementos dos vetores `x` e `y`, elemento por elemento. A saída deve ser outro vetor de mesmo tamanho e com elementos iguais a operação `x + y`. Use a função `identical` para testar se todos os elementos de ambos os vetores são iguais.

03 - Use o pacote `tictoc` para comparar o desempenho da função anterior com o operador nativo `+`, e uma versão baseada em loops com pré-alocação. Qual alternativa apresenta menor tempo de execução e por quê? A versão `Rcpp` vence a versão em loop?

04 - Use o pacote `memoise` para criar uma versão memorizada de `Quandl::Quandl`. Use a nova função para importar dados para o Índice de Preços ao Consumidor dos Estados Unidos (código `'FRED/DDOE01USA086NWDB'`). Quanto de ganho de velocidade em porcentagem você obtém da segunda chamada para a versão memorizada?

# Referências Bibliográficas

- Armstrong, W., Eddelbuettel, D., and Laing, J. (2022). *Rblpapi: R Interface to Bloomberg*. R package version 0.3.14.
- Bache, S. M. and Wickham, H. (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.3.
- Dahl, D. B., Scott, D., Roosen, C., Magnusson, A., and Swinton, J. (2019). *xtable: Export Tables to LaTeX or HTML*. R package version 1.8-4.
- Dowle, M. and Srinivasan, A. (2022). *data.table: Extension of 'data.frame'*. R package version 1.14.6.
- Dragulescu, A. and Arendt, C. (2020). *xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*. R package version 0.6.5.
- Ferreira, P. C., Speranza, T., and Costa, J. (2018). *BETS: Brazilian Economic Time Series*. R package version 0.4.9.
- Freitas, W. (2022). *bizdays: Business Days Calculations and Utilities*. R package version 1.0.12.
- Garmonsway, D. (2022). *tidyxl: Read Untidy Excel Files*. R package version 1.0.8.
- Gentzkow, M., Kelly, B. T., and Taddy, M. (2017). Text as data. Technical report, National Bureau of Economic Research.
- Gomolka, M. (2021). *simfinapi: Accessing SimFin Data*. R package version 0.2.0.
- James, D. and Hornik, K. (2022). *chron: Chronological Objects which can Handle Dates and Times*. R package version 2.3-58.
- Leifeld, P. (2022). *texreg: Conversion of R Regression Output to LaTeX or HTML Tables*. R package version 1.38.6.

- McLeish, D. L. (2011). *Monte Carlo simulation and finance*, volume 276. John Wiley & Sons.
- Mirai Solutions GmbH (2022). *XLConnect: Excel Connector for R*. R package version 1.0.6.
- Ooms, J. (2022). *writexl: Export Data Frames to Excel xlsx Format*. R package version 1.4.1.
- Perlin, M. (2021). *GetDFPData: Reading Annual Financial Reports from Bovespa's DFP, FRE and FCA System*. R package version 1.6.
- Perlin, M. (2022a). *BatchGetSymbols: Downloads and Organizes Financial Data for Multiple Tickers*. R package version 2.6.4.
- Perlin, M. (2022b). *GetBCBData: Imports Datasets from BCB (Central Bank of Brazil) using Its Official API*. R package version 0.7.0.
- Perlin, M. (2022c). *GetTDDData: Get Data for Brazilian Bonds (Tesouro Direto)*. R package version 1.5.1.
- Perlin, M. and Kirch, G. (2022a). *GetDFPData2: Reading Annual and Quarterly Financial Reports from B3*. R package version 0.6.2.
- Perlin, M. and Kirch, G. (2022b). *GetFREDData: Reading FRE Corporate Data of Public Traded Companies from B3*. R package version 0.8.1.
- Perlin, M., Kirch, G., and Vancin, D. (2018). Accessing financial reports and corporate events with getdfpdata. Available at SSRN 3128252.
- Perlin, M. S. (2019). *GetQuandlData: Fast and Cached Import of Data from Quandl Using the json API*. R package version 0.1.0.
- R Core Team (2022). *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...* R package version 0.8-82.
- Raymond McTaggart, Gergely Daroczi, and Clement Leung (2021). *Quandl: API Wrapper for Quandl.com*. R package version 2.11.0.
- Ryan, J. A. and Ulrich, J. M. (2022a). *IBrokers: R API to Interactive Brokers Trader Workstation*. R package version 0.10-2.
- Ryan, J. A. and Ulrich, J. M. (2022b). *xts: eXtensible Time Series*. R package version 0.12.2.
- Spinu, V., Grolemond, G., and Wickham, H. (2022). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.9.0.
- Teetor, P. (2011). *R cookbook*. "O'Reilly Media, Inc."

- Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.
- Vaughan, D. and Dancho, M. (2022). *tibbletime: Time Aware Tibbles*. R package version 0.1.7.
- Wickham, H. (2019). *Advanced r*. CRC press.
- Wickham, H. (2022a). *rvest: Easily Harvest (Scrape) Web Pages*. R package version 1.0.3.
- Wickham, H. (2022b). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.1.
- Wickham, H. and Bryan, J. (2022). *readxl: Read Excel Files*. R package version 1.4.1.
- Wuertz, D., Setz, T., and Chalabi, Y. (2022). *timeDate: Rmetrics - Chronological and Calendar Objects*. R package version 4021.106.
- Xie, Y. (2022). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.30.
- Zeileis, A., Gruen, B., Leisch, F., and Umlauf, N. (2022). *exams: Automatic Generation of Exams in R*. R package version 2.4-0.



# Índice Remissivo

- base
  - ?, 69
  - as.character, 214
  - browser, 47
  - c, 49
  - class, 49, 51
  - colnames, 183
  - cut, 201
  - data.frame, 153
  - dim, 56
  - dirname, 65
  - expand.grid, 213
  - factor, 217
  - format, 54, 235
  - getwd, 63, 90
  - is.na, 240
  - length, 56
  - list, 173
  - load, 104
  - ls, 50
  - merge, 167
  - message, 52
  - na.omit, 241
  - ncol, 56
  - nrow, 56
  - order, 162
  - outer, 213
  - paste, 54
  - paste0, 54
  - print, 37
  - rep, 193
  - rm, 62
  - rnorn, 194
  - rownames, 183
  - runif, 194
  - sample, 196
  - seq, 192, 228
  - set.seed, 197
  - source, 45
  - str, 51
  - Sys.Date, 125, 236
  - Sys.time, 236
  - which, 222
- BatchGetSymbols, 124
  - BatchGetSymbols, 125
  - GetIbovStocks, 127
- bizdays, 225
- caracteres latinos, 32
- chron, 225
- constants
  - LETTERS, 207
  - letters, 207
  - month.abb, 207
  - month.name, 207
- devtools, 40

- install\_github, 40
- dplyr
  - %>%, 156
  - arrange, 162
  - bind\_cols, 165
  - bind\_rows, 165
  - desc, 164
  - filter, 161
  - select, 159
- forcats
  - fct\_recode, 219
- fst, 106
- GetDFPData2, 137
  - export\_xlsx, 141
  - get.info.companies, 137
  - get\_dfp\_data, 139
  - search\_company, 139
- GetQuandlData, 118
- GetTDDData, 128
  - download.TD.data, 128
  - get.yield.curve, 132
  - read.TD.files, 128
- lubridate
  - dmy, 226
  - hour, 235
  - mdy, 226
  - now, 236
  - today, 236
  - ymd, 226
- MariaDB, 109
- mySQL, 109
- Quandl, 118
- readr, 94
  - cols, 95
  - read\_csv, 94
  - write\_csv, 97
- readxl, 99
  - read\_excel, 99
- RSQLite, 110
- dbConnect, 109, 111
- dbDisconnect, 111
- dbGetQuery, 109
- dbReadTable, 109
- dbWriteTable, 111
- rstudioapi
  - getActiveDocumentContext, 65
- rvest, 146
  - read\_html, 146
  - html\_nodes, 146
  - html\_table, 146
- SQLite, 109
- stringr, 205
  - fixed, 209
  - locate, 209
  - locate\_all, 209
  - str\_c, 206
  - str\_dup, 207
  - str\_length, 213
  - str\_replace, 210
  - str\_replace\_all, 210
  - str\_split, 212
  - str\_sub, 208
- Task Views, 18
- tibble, 153
  - data\_frame, 153
  - tibble, 153
- tidyxl, 99
- timeDate, 225
- utils
  - install.packages, 39
  - update.packages, 43
  - View, 154
  - zip, 114
- writexl
  - write\_xlsx, 101
- XLConnet, 99
- xlsx, 99, 100
  - write.xlsx, 101