DATA SCIENCE SERIES

INTRODUCTION TO DATA SCIENCE

Data Wrangling and Visualization with R

SECOND EDITION

RAFAEL A. IRIZARRY

A Chapman & Hall Book

FREE INSTRUCTOR & STUDENT RESOURCES

CRC Press
Taylor & Francis Group

Introduction to Data Science

Unlike the first edition, the new edition has been split into two books.

Thoroughly revised and updated, this is the first book of the second edition of Introduction to Data Science: Data Wrangling and Visualization with R. It introduces skills that can help you tackle real-world data analysis challenges. These include R programming, data wrangling with dplyr, data visualization with ggplot2, file organization with UNIX/Linux shell, version control with Git and GitHub, and reproducible document preparation with Quarto and knitr. The new edition includes additional material/chapters on data.table, locales, and accessing data through APIs. The book is divided into four parts: R, Data Visualization, Data Wrangling, and Productivity Tools. Each part has several chapters meant to be presented as one lecture and includes dozens of exercises. The second book will cover topics including probability, statistics and prediction algorithms with R.

Throughout the book, we use motivating case studies. In each case study, we try to realistically mimic a data scientist's experience. For each of the skills covered, we start by asking specific questions and answer these through data analysis. Examples of the case studies included in the book are: US murder rates by state, self-reported student heights, trends in world health and economics, and the impact of vaccines on infectious disease rates.

This book is meant to be a textbook for a first course in Data Science. No previous knowledge of R is necessary, although some experience with programming may be helpful. To be a successful data analyst implementing these skills covered in this book requires understanding advanced statistical concepts, such as those covered the second book. If you read and understand all the chapters and complete all the exercises in this book, and understand statistical concepts, you will be well-positioned to perform basic data analysis tasks and you will be prepared to learn the more advanced concepts and skills needed to become an expert.

Rafael A. Irizarry is professor and chair of Data Science at the Dana-Farber Cancer Institute, professor of biostatistics at Harvard, and a fellow of the American Statistical Association and the International Society of Computational Biology. Prof. Irizarry is an applied statistician and during the last 25 years has worked in diverse areas, including genomics, sound engineering, and public health surveillance. He disseminates solutions to data analysis challenges as open source software, tools that are widely downloaded and used. Prof. Irizarry has also developed and taught several data science courses at Harvard as well as popular online courses.

CHAPMAN & HALL/CRC DATA SCIENCE SERIES

Reflecting the interdisciplinary nature of the field, this book series brings together researchers, practitioners, and instructors from statistics, computer science, machine learning, and analytics. The series will publish cutting-edge research, industry applications, and textbooks in data science.

The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes titles in the areas of machine learning, pattern recognition, predictive analytics, business analytics, Big Data, visualization, programming, software, learning analytics, data wrangling, interactive graphics, and reproducible research.

Recently Published Titles

Practitioner's Guide to Data Science
Hui Lin and Ming Li

Natural Language Processing in the Real World
Text Processing, Analytics, and Classification
Jyotika Singh

Telling Stories with Data
With Applications in R
Rohan Alexander

Big Data Analytics
A Guide to Data Science Practitioners Making the Transition to Big Data
Ulrich Matter

Data Science for Sensory and Consumer Scientists
Thierry Worch, Julien Delarue, Vanessa Rios De Souza and John Ennis

Data Science in Practice
Tom Alby

Introduction to NFL Analytics with R
Bradley J. Congelio

Soccer Analytics
An Introduction Using R
Clive Beggs

Spatial Statistics for Data Science
Theory and Practice with R
Paula Moraga

Research Software Engineering
A Guide to the Open Source Ecosystem
Matthias Bannert

The Data Preparation Journey
Finding Your Way With R
Martin Hugh Monkman

Getting (more out of) Graphics
Practice and Principles of Data Visualisation
Antony Unwin

Introduction to Data Science
Data Wrangling and Visualization with R Second Edition
Rafael A. Irizarry

For more information about this series, please visit: https://www.routledge.com/Chapman--HallCRC-Data-Science-Series/book-series/CHDSS

Introduction to Data Science

Data Wrangling and Visualization with R

Second Edition

Rafael A. Irizarry



CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Designed cover image: © Rafael A. Irizarry

First edition published 2020
Second edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2025 Rafael A. Irizarry

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 978-1-032-11655-6 (hbk)
ISBN: 978-1-032-11654-9 (pbk)
ISBN: 978-1-003-22092-3 (ebk)

DOI: 10.1201/9781003220923

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

Access the Instructor and Student Resources: https://www.routledge.com/Introduction-to-Data-Science-Data-Wrangling-and-Visualization-with-R/Irizarry/p/book/9781032116556

Table of contents

Preface **xiii**

Acknowledgments **xv**

Introduction **xvii**
Case studies . . . xvii
Who will find this book useful? . . . xvii
What does this book cover? . . . xviii
What is not covered by this book? . . . xviii

I R **1**

1 Getting started **4**
1.1 Why R? . . . 4
1.2 The R console . . . 4
1.3 Scripts . . . 5
1.4 RStudio . . . 6
1.4.1 The panes . . . 6
1.4.2 Key bindings . . . 7
1.4.3 Running commands while editing scripts . . . 8
1.4.4 Changing global options . . . 10
1.5 Installing R packages . . . 10

2 R basics **12**
2.1 Motivating example: US Gun Murders . . . 12
2.2 The very basics . . . 14
2.2.1 Objects . . . 14
2.2.2 The workspace . . . 14
2.2.3 Prebuilt functions . . . 15
2.2.4 Prebuilt objects . . . 17
2.2.5 Variable names . . . 17
2.2.6 Saving your workspace . . . 18
2.2.7 Why use scripts? . . . 18
2.2.8 Commenting your code . . . 18
2.3 Data types . . . 19
2.3.1 Data frames . . . 19
2.3.2 Examining objects . . . 20
2.3.3 The accessor: $ . . . 20
2.3.4 Vectors . . . 21
2.3.5 Factors . . . 22

2.3.6 Lists . . . 23
2.3.7 Matrices . . . 24
2.4 Vectors . . . 26
2.4.1 Creating vectors . . . 26
2.4.2 Names . . . 27
2.4.3 Sequences . . . 27
2.4.4 Subsetting . . . 28
2.5 Coercion . . . 29
2.6 Not availables (NA) . . . 30
2.7 Sorting . . . 30
2.7.1 `sort` . . . 30
2.7.2 `order` . . . 31
2.7.3 `max` and `which.max` . . . 32
2.7.4 `rank` . . . 32
2.8 Vector arithmetics . . . 32
2.8.1 Rescaling a vector . . . 33
2.8.2 Two vectors . . . 33
2.8.3 Beware of recycling . . . 34
2.9 Indexing . . . 34
2.9.1 Subsetting with logicals . . . 34
2.9.2 Logical operators . . . 35
2.9.3 `which` . . . 36
2.9.4 `match` . . . 36
2.9.5 `%in%` . . . 36
2.10 Basic plots . . . 37
2.10.1 `plot` . . . 37
2.10.2 `hist` . . . 37
2.10.3 `boxplot` . . . 38
2.10.4 `image` . . . 38
2.11 Exercises . . . 39

3 Programming basics 44
3.1 Conditional expressions . . . 44
3.2 Defining functions . . . 46
3.3 Namespaces . . . 47
3.4 For-loops . . . 48
3.5 Vectorization and functionals . . . 49
3.6 Exercises . . . 51

4 The tidyverse 52
4.1 Tidy data . . . 52
4.2 Refining data frames . . . 53
4.2.1 Adding columns . . . 53
4.2.2 Row-wise subsetting . . . 54
4.2.3 Column-wise subsetting . . . 54
4.2.4 Transforming variables . . . 55
4.3 The pipe . . . 56
4.4 Summarizing data . . . 57
4.4.1 The `summarize` function . . . 57
4.4.2 Multiple summaries . . . 58
4.4.3 Group then summarize with `group_by` . . . 59
4.4.4 Extracting variables with `pull` . . . 60

4.5 Sorting . . . 60
4.5.1 Nested sorting . . . 61
4.5.2 The top n . . . 61
4.6 Tibbles . . . 62
4.7 Tibbles versus data frames . . . 63
4.7.1 Creating tibbles . . . 64
4.8 The placeholder . . . 64
4.9 The **purrr** package . . . 65
4.10 Tidyverse conditionals . . . 66
4.10.1 `case_when` . . . 66
4.10.2 `between` . . . 67
4.11 Exercises . . . 67

5 data.table **72**
5.1 Refining data tables . . . 72
5.1.1 Column-wise subsetting . . . 72
5.1.2 Adding or transformin variables . . . 73
5.1.3 Reference versus copy . . . 73
5.1.4 Row-wise subsetting . . . 74
5.2 Summarizing data . . . 75
5.2.1 Multiple summaries . . . 75
5.2.2 Group then summarize . . . 76
5.3 Sorting . . . 76
5.4 Exercises . . . 77

6 Importing data **79**
6.1 Navigating and managing the filesystem . . . 80
6.1.1 The filesystem . . . 80
6.1.2 Relative and full paths . . . 80
6.1.3 The working directory . . . 81
6.1.4 Generating path names . . . 81
6.2 File types . . . 82
6.2.1 Text files . . . 82
6.2.2 Binary files . . . 82
6.2.3 Encoding . . . 83
6.3 Parsers . . . 83
6.3.1 Base R . . . 83
6.3.2 readr . . . 84
6.3.3 readxl . . . 85
6.3.4 data.table . . . 86
6.3.5 Downloading files . . . 86
6.4 Organizing data with spreadsheets . . . 87
6.5 Exercises . . . 87

II Data Visualization **89**

7 Visualizing data distributions **95**
7.1 Variable types . . . 95
7.2 Case study: describing student heights . . . 96
7.3 Distributions . . . 96

7.3.1 Histograms . . . 97
7.3.2 Smoothed density . . . 99
7.3.3 The normal distribution . . . 100
7.4 Boxplots . . . 102
7.5 Stratification . . . 103
7.6 Case study continued . . . 104
7.7 Exercises . . . 105

8 ggplot2 107
8.1 The components of a graph . . . 108
8.2 Initializing an object with data . . . 109
8.3 Adding a geometry . . . 110
8.4 Aesthetic mappings . . . 111
8.5 Other layers . . . 112
8.6 Global aesthetic mappings . . . 112
8.7 Non-aesthetic arguments . . . 113
8.8 Categories as colors . . . 114
8.9 Updating ggplot objects . . . 115
8.10 Scales . . . 116
8.11 Annotations . . . 116
8.12 Add-on packages . . . 118
8.13 Putting it all together . . . 119
8.14 Geometries . . . 119
8.14.1 Barplots . . . 120
8.14.2 Histograms . . . 120
8.14.3 Density plots . . . 120
8.14.4 Boxplots . . . 121
8.14.5 Images . . . 121
8.15 Grids of plots . . . 122
8.16 Exercises . . . 122

9 Data visualization principles 126
9.1 Encoding data using visual cues . . . 126
9.2 Know when to include 0 . . . 129
9.3 Do not distort quantities . . . 133
9.4 Order categories by a meaningful value . . . 135
9.5 Show the data . . . 136
9.6 Ease comparisons . . . 138
9.6.1 Use common axes . . . 138
9.6.2 Aligning plots for comparisons . . . 139
9.7 Transformations . . . 141
9.8 Visual cues to be compared should be adjacent . . . 143
9.9 Think of the color blind . . . 144
9.10 Plots for two variables . . . 144
9.10.1 Slope charts . . . 145
9.10.2 Bland-Altman plot . . . 146
9.11 Encoding a third variable . . . 146
9.12 Avoid pseudo-three-dimensional plots . . . 148
9.13 Avoid too many significant digits . . . 149
9.14 Know your audience . . . 150
9.15 Exercises . . . 150

10 Data visualization in practice **155**
10.1 Case study 1: new insights on poverty . . . 155
10.2 Scatterplots . . . 157
10.3 Faceting . . . 158
10.3.1 `facet_wrap` . . . 160
10.3.2 Fixed scales for better comparisons . . . 160
10.4 Time series plots . . . 161
10.5 Data transformations . . . 165
10.5.1 Log transformation . . . 166
10.5.2 Which base? . . . 167
10.5.3 Transform the values or the scale? . . . 168
10.6 Multimodal distributions . . . 169
10.7 Comparing distributions . . . 169
10.7.1 Boxplots . . . 171
10.7.2 Ridge plots . . . 172
10.7.3 Example: 1970 versus 2010 income distributions . . . 174
10.7.4 Accessing computed variables . . . 178
10.7.5 Weighted densities . . . 180
10.8 Case study 2: the ecological fallacy . . . 181
10.8.1 Logistic transformation . . . 182
10.8.2 Show the data . . . 182
10.9 Case study 3: vaccines and infectious diseases . . . 183
10.9.1 Vaccine data . . . 183
10.9.2 Trend plots . . . 184
10.10 Heatmaps . . . 184
10.11 Exercises . . . 187

III Data Wrangling **189**

11 Reshaping data **192**
11.1 `pivot_longer` . . . 192
11.2 `pivot_wider` . . . 193
11.3 Separating variables . . . 194
11.4 Reshaping with data.table . . . 196
11.4.1 `pivot_longer` is `melt` . . . 196
11.4.2 `pivot_wider` is `dcast` . . . 197
11.4.3 Separating variables . . . 197
11.5 The janitor package . . . 198
11.6 Exercises . . . 199

12 Joining tables **201**
12.1 Joins . . . 202
12.1.1 Left join . . . 203
12.1.2 Right join . . . 203
12.1.3 Inner join . . . 203
12.1.4 Full join . . . 204
12.1.5 Semi join . . . 204
12.1.6 Anti join . . . 204
12.2 Binding . . . 205

12.2.1 Binding columns . . . 205
12.2.2 Binding by rows . . . 206
12.3 Set operators . . . 206
12.3.1 Intersect . . . 206
12.3.2 Union . . . 207
12.3.3 `setdiff` . . . 207
12.3.4 `setequal` . . . 208
12.4 Joining with data.table . . . 208
12.5 Exercises . . . 209

13 Parsing dates and times **210**
13.1 The date data type . . . 210
13.2 The lubridate package . . . 211
13.3 Dates and times with data.table . . . 214
13.4 Exercises . . . 215

14 Locales **216**
14.1 Locales in R . . . 216
14.2 The `locale` function . . . 217
14.3 Example: wrangling a Spanish dataset . . . 218
14.4 Exercises . . . 220

15 Extracting data from the web **221**
15.1 Scraping HTML . . . 221
15.1.1 HTML . . . 222
15.1.2 The rvest package . . . 223
15.1.3 CSS selectors . . . 225
15.2 JSON . . . 226
15.3 Data APIs . . . 227
15.3.1 API types and concepts . . . 227
15.3.2 The httr2 package . . . 228
15.4 Exercises . . . 229

16 String processing **231**
16.1 The stringr package . . . 231
16.2 Case study 1: self-reported heights . . . 233
16.3 Escaping . . . 235
16.4 Regular expressions . . . 237
16.4.1 Strings are a regex . . . 237
16.4.2 Special characters . . . 237
16.4.3 Character classes . . . 239
16.4.4 Anchors . . . 240
16.4.5 Bounded quantifiers . . . 240
16.4.6 White space . . . 241
16.4.7 Unbounded quantifiers: `*`, `?`, `+` . . . 242
16.4.8 Not . . . 243
16.4.9 Groups . . . 243
16.4.10 Search and replace . . . 244
16.4.11 Search and replace using groups . . . 246
16.4.12 Lookarounds . . . 246
16.4.13 Separating variables . . . 247

16.5 Trimming . . . 248
16.6 Case conversion . . . 248
16.7 Case study 1 continued: Putting it all together . . . 248
16.8 Case study 2: extracting tables from a PDF . . . 252
16.9 Renaming levels . . . 256
16.10 Exercises . . . 257

17 Text analysis **260**
17.1 Case study: Trump tweets . . . 260
17.2 Text as data . . . 262
17.3 Sentiment analysis . . . 266
17.4 Exercises . . . 270

IV Productivity Tools **271**

18 Organizing with Unix **274**
18.1 Naming convention . . . 274
18.2 The terminal . . . 275
18.3 The filesystem . . . 275
18.3.1 Directories and subdirectories . . . 276
18.3.2 The home directory . . . 276
18.3.3 Working directory . . . 278
18.3.4 Paths . . . 278
18.4 Unix commands . . . 279
18.4.1 `ls`: Listing directory content . . . 279
18.4.2 `mkdir` and `rmdir`: make and remove a directory . . . 279
18.4.3 `cd`: navigating the filesystem by changing directories . . . 280
18.4.4 Examples . . . 281
18.5 More Unix commands . . . 283
18.5.1 `mv`: moving files . . . 283
18.5.2 `cp`: copying files . . . 284
18.5.3 `rm`: removing files . . . 284
18.5.4 `less`: looking at a file . . . 284
18.6 Case study: Preparing for a project . . . 285
18.7 Advanced Unix . . . 285
18.7.1 Arguments . . . 286
18.7.2 Getting help . . . 286
18.7.3 Pipes . . . 287
18.7.4 Wild cards . . . 287
18.7.5 Environment variables . . . 288
18.7.6 Shells . . . 288
18.7.7 Executables . . . 289
18.7.8 Permissions and file types . . . 289
18.7.9 Commands you should learn . . . 289
18.7.10 File manipulation in R . . . 290

19 Git and GitHub **291**
19.1 Why use Git and GitHub? . . . 291
19.2 Overview of Git . . . 292
19.3 GitHub accounts . . . 293

19.4 GitHub repositories . . . 293
19.5 Connecting Git and GitHub . . . 294
19.6 Initial setup . . . 295
19.7 Git basics . . . 296
19.7.1 The working directory . . . 296
19.7.2 add . . . 297
19.7.3 commit . . . 298
19.7.4 push . . . 299
19.7.5 fetch and merge . . . 300
19.7.6 pull . . . 300
19.7.7 clone . . . 300
19.8 .gitignore . . . 301
19.9 Git in RStudio . . . 301

20 Reproducible projects **306**
20.1 RStudio projects . . . 306
20.2 Markdown . . . 309
20.2.1 The header . . . 310
20.2.2 R code chunks . . . 311
20.2.3 Global execution options . . . 311
20.2.4 knitR . . . 312
20.2.5 Learning more . . . 313
20.3 Organizing a data science project . . . 313
20.3.1 Create directories in Unix . . . 313
20.3.2 Create an RStudio project . . . 313
20.3.3 Editting R scripts . . . 314
20.3.4 Saving processed data . . . 315
20.3.5 The main analysis file . . . 315
20.3.6 Other directories . . . 316
20.3.7 The README file . . . 316
20.3.8 Initializing a Git directory . . . 316
20.3.9 Add, commit, and push files using RStudio . . . 317

Index **319**

Preface

We make announcements related to the book on Twitter. For updates follow @rafalab[1].

This book started out as the class notes used in the HarvardX Data Science Series[2].

The Quarto files used to generate the book is available on GitHub[3]. Note that, the graphical theme used for plots throughout the book can be recreated using the `ds_theme_set()` function from **dslabs** package.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International CC BY-NC-SA 4.0[4].

A free PDF of the October 24, 2019 version of the book is available from Leanpub[5].

[1] https://twitter.com/rafalab
[2] https://www.edx.org/professional-certificate/harvardx-data-science
[3] https://github.com/rafalab/dsbook-part-1
[4] https://creativecommons.org/licenses/by-nc-sa/4.0
[5] https://leanpub.com/datasciencebook



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Acknowledgments

This book is dedicated to all the people involved in building and maintaining R and the R packages we use in this book. A special thanks to the developers and maintainers of base R, the tidyverse, data.table, and the caret package.

A special thanks to Jenna Landy for her careful editing and helpful advice on this book; to David Robinson for generously answering many questions about the tidyverse and aiding in my understanding of it; and to Amy Gill for dozens of comments, edits, and suggestions. Also, many thanks to Stephanie Hicks who twice served as a co-instructor in my data science classes and Yihui Xie who patiently put up with my many questions related to markdown. Thanks also to Karl Broman, from whom I borrowed ideas for the Data Visualization and Productivity Tools parts. Thanks to Peter Aldhous from whom I borrowed ideas for the principles of data visualization section and Jenny Bryan for writing *Happy Git and GitHub for the useR*, which influenced our Git chapters. Also, many thanks to Jeff Leek, Roger Peng, and Brian Caffo, whose online classes inspired the way this book is divided, to Garrett Grolemund and Hadley Wickham for making the markdown code for their R for Data Science book open, and the editors, John Kimmel and Lara Spieker, for their support. Finally, thanks to Alex Nones for proofreading the manuscript during its various stages.

This book was conceived during the teaching of several applied statistics courses, starting over twenty years ago. The teaching assistants working with me throughout the years made important indirect contributions to this book. The material was further refined during a HarvardX series coordinated by Heather Sternshein and Zofia Gajdos. We thank them for their contributions. We are also grateful to all the students whose questions and comments helped us improve the book. The courses were partially funded by NIH grant R25GM114818. We are very grateful to the National Institutes of Health for its support.

A special thanks goes to all those who edited the book via GitHub pull requests or made suggestions by creating an *issue* or sending an email: `nickyfoto` (Huang Qiang), `desautm` (Marc-André Désautels), `michaschwab` (Michail Schwab), `alvarolarreategui` (Alvaro Larreategui), `jakevc` (Jake VanCampen), `omerta` (Guillermo Lengemann), `espinielli` (Enrico Spinielli), `asimumba`(Aaron Simumba), `braunschweig` (Maldewar), `gwierzchowski` (Grzegorz Wierzchowski), `technocrat` (Richard Careaga), `atzakas`, `defeit` (David Emerson Feit), `shiraamitchell` (Shira Mitchell), `Nathalie-S`, `andreashandel` (Andreas Handel), `berkowitze` (Elias Berkowitz), `Dean-Webb` (Dean Webber), `mohayusuf`, `jimrothstein`, `mPloenzke` (Matthew Ploenzke), `NicholasDowand` (Nicholas Dow), `kant` (Darío Hereñú), `debbieyuster` (Debbie Yuster), `tuanchauict` (Tuan Chau), `phzeller`, `BTJ01` (BradJ), `glsnow` (Greg Snow), `mberlanda` (Mauro Berlanda), `wfan9`, `larswestvang` (Lars Westvang), `jj999` (Jan Andrejkovic), `Kriegslustig` (Luca Nils Schmid), `odahhani`, `aidanhorn` (Aidan Horn), `atraxler` (Adrienne Traxler), `alvegorova`,`wycheong` (Won Young Cheong), `med-hat` (Medhat Khalil), `kengustafson`, `Yowza63`, `ryan-heslin` (Ryan Heslin), `raffaem`, `tim8west`, `jooleer`, `pauluhn` (Paul), `tci1`, `beanb2` (Brennan Bean), `edasdemirlab` (Erdi Dasdemir), David D. Kane, El Mustapha El Abbassi, Vadim Zipunnikov, Anna Quaglieri, Chris Dong, Bowen Gu, and Rick Schoenberg.



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Introduction

The demand for skilled data science practitioners in industry, academia, and government is rapidly growing. This book introduces skills that can help you tackle real-world data analysis challenges. These include R programming, data wrangling with **dplyr**, data visualization with **ggplot2**, file organization with Unix/Linux shell, version control with Git and GitHub, and reproducible document preparation with Quarto and **knitr**. The book is divided into four parts: **R**, **Data Visualization**, **Data Wrangling**, and **Productivity Tools**. Each part has several chapters meant to be presented as one lecture and includes dozens of exercises distributed across chapters.

Case studies

Throughout the book, we use motivating case studies. In each case study, we try to realistically mimic a data scientist's experience. For each of the skills covered, we start by asking specific questions and answer these through data analysis. We learn the concepts as a means to answer the questions. Examples of the case studies included in the book are:

Case Study	Concept
US murder rates by state	R Basics
Student heights	Statistical Summaries
Trends in world health and economics	Data Visualization
The impact of vaccines on infectious disease rates	Data Visualization
Reported student heights	Data Wrangling

Who will find this book useful?

This book is meant to be a textbook for a first course in Data Science. No previous knowledge of R is necessary, although some experience with programming may be helpful. To be a successful data analyst implementing these skill requires understanding advanced statistical concepts, such as those covered in *the second book*. If you read and understand all the chapters and complete all the exercises in this book, and understand statistical concepts, you will be well-positioned to perform basic data analysis tasks and you will be prepared to learn the more advanced concepts and skills needed to become an expert.

What does this book cover?

We start by going over the **basics of R**, the **tidyverse**, and the **data.table** package. You learn R throughout the book, but in the first part we go over the building blocks needed to keep learning.

The growing availability of informative datasets and software tools has led to increased reliance on **data visualizations** in many fields. In the second part we demonstrate how to use **ggplot2** to generate graphs and describe important data visualization principles.

The third part uses several examples to familiarize the reader with **data wrangling**. Among the specific skills we learn are web scraping, using regular expressions, and joining and reshaping data tables. We do this using **tidyverse** tools.

In the final part, we provide a brief introduction to the **productivity tools** we use on a day-to-day basis in data science projects. These are RStudio, Unix/Linux shell, Git and GitHub, Quarto, and **knitr**.

What is not covered by this book?

This book focuses on the computing skills necessary for the data analysis aspects of data science. As mentioned, we do not cover statistical concepts. We also do not cover aspects related to data management or engineering. Although R programming is an essential part of the book, we do not teach more advanced computer science topics such as data structures, optimization, and algorithm theory. Similarly, we do not cover topics such as web services, interactive graphics, parallel computing, and data streaming processing.

Part I

R



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

In this book, we will be using the R software environment for all our analysis. You will learn R and data analysis techniques simultaneously. To follow along you will therefore need access to R. We also recommend the use of an *integrated development environment* (IDE), such as RStudio, to save your work. Note that it is common for a course or workshop to offer access to an R environment and an IDE through your web browser, as done by RStudio cloud[6]. If you have access to such a resource, you don't need to install R and RStudio. However, if you intend on becoming an advanced data analyst, we highly recommend installing these tools on your computer. Both R and RStudio are free and available online.

[6] https://rstudio.cloud

1

Getting started

1.1 Why R?

R is not a programming language like C or Java. It was not created by software engineers for software development. Instead, it was developed by statisticians as an interactive environment for data analysis. You can read the full history in the paper A Brief History of S[1]. The interactivity is an indispensable feature in data science because, as you will soon learn, the ability to quickly explore data is a necessity for success in this field. However, like in other programming languages, you can save your work as scripts that can be easily executed at any moment. These scripts serve as a record of the analysis you performed, a key feature that facilitates reproducible work. If you are an expert programmer, you should not expect R to follow the conventions you are used to since you will be disappointed. If you are patient, you will come to appreciate the unequal power of R when it comes to data analysis and, specifically, data visualization.

Other attractive features of R are:

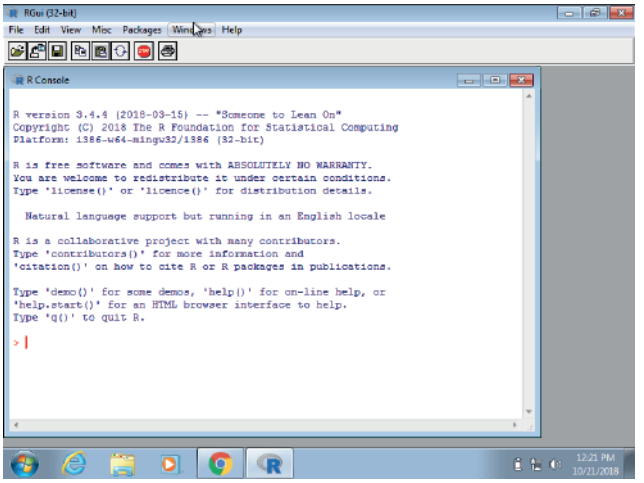
1. R is free and open source[2].
2. It runs on all major platforms: Windows, macOS, Unix/Linux.
3. Scripts and data objects can be shared seamlessly across platforms.
4. There is a large, growing, and active community of R users, and, as a result, there are numerous resources for learning and asking questions[3,4].
5. It is easy for others to contribute add-ons, which enables developers to share software implementations of new data science methodologies. This gives R users early access to the latest methods and tools which are developed for a wide variety of disciplines, including ecology, molecular biology, social sciences, and geography, just to name a few examples.

1.2 The R console

Interactive data analysis usually occurs on the *R console* that executes commands as you type them. There are several ways to gain access to an R console. One way is to simply start R on your computer. The console looks something like this:

[1] https://pdfs.semanticscholar.org/9b48/46f192aa37ca122cfabb1ed1b59866d8bfda.pdf
[2] https://opensource.org/history
[3] https://stats.stackexchange.com/questions/138/free-resources-for-learning-r
[4] https://www.r-project.org/help.html

DOI: 10.1201/9781003220923-1



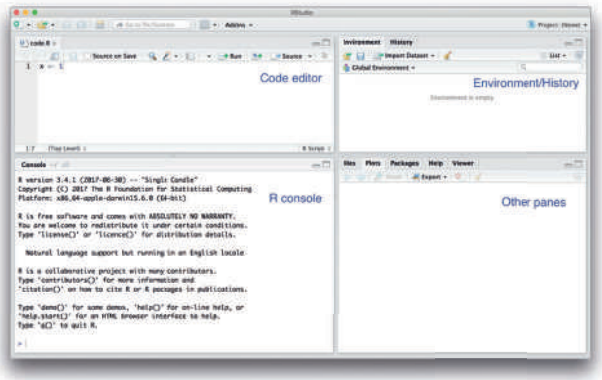
As a quick example, try using the console to calculate a 15% tip on a meal that costs $19.71:

```
0.15 * 19.71
#> [1] 2.96
```

Note that in this book, gray sidebars are used to show R code typed into the R console. The symbol #> is used to denote what the R console outputs.

1.3 Scripts

One of the great advantages of R over point-and-click analysis software is that you can save your work as scripts. You can edit and save these scripts using a text editor. The material in this book was developed using the interactive *integrated development environment* (IDE) RStudio[5]. RStudio includes an editor with many R-specific features, a console to execute your code, and other useful panes, including one to show figures.



[5] https://posit.co//

Most web-based R consoles also provide a pane to edit scripts, but not all permit you to save the scripts for later use.

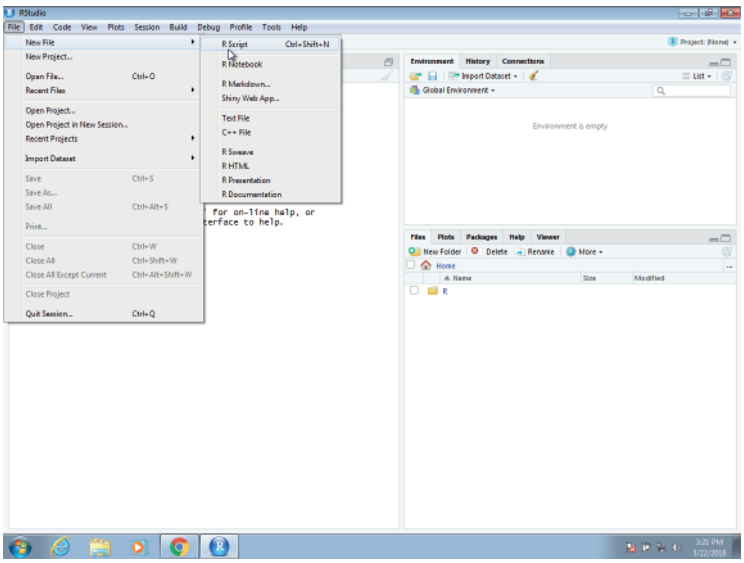
All the R scripts used to generate this book can be found on GitHub[6].

1.4 RStudio

RStudio will be our launching pad for data science projects. It provides not only an editor for us to create and edit our scripts but also many other useful tools. In this section, we go over some of the basics.

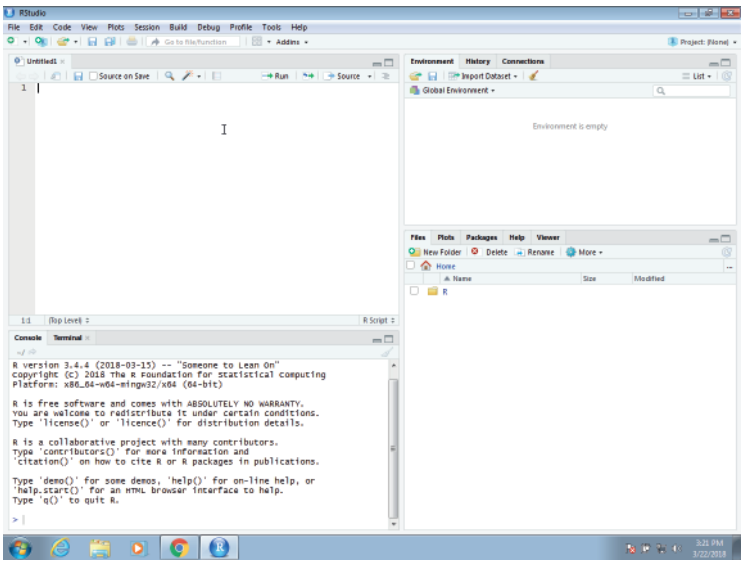
1.4.1 The panes

When you start RStudio for the first time, you will see three panes. The left pane shows the R console. On the right, the top pane includes tabs such as *Environment* and *History*, while the bottom pane shows five tabs: *File*, *Plots*, *Packages*, *Help*, and *Viewer* (these tabs may change in new versions). You can click on each tab to move across the different features. For example, to start a new script, you can click on File, then New File, then R Script.



This starts a new pane on the left and it is here where you can start writing your script.

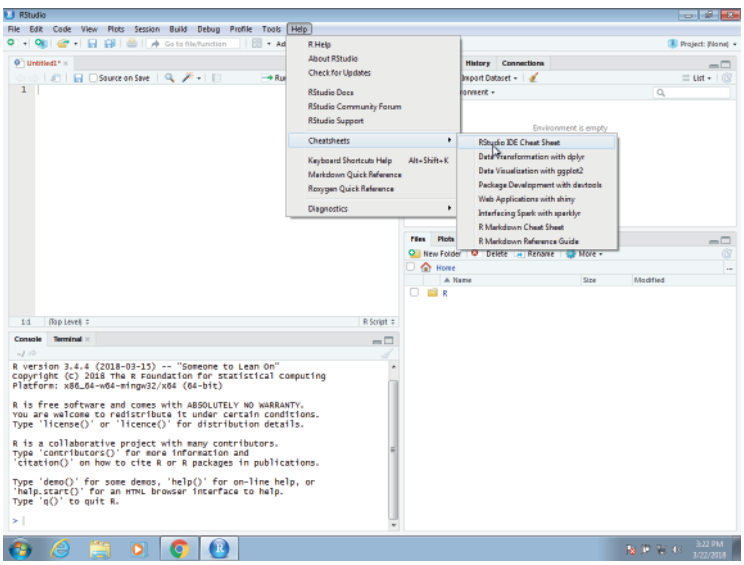
[6] https://github.com/rafalab/dsbook-part-1



1.4.2 Key bindings

Many tasks we perform with the mouse can be achieved with a combination of keystrokes instead. These keyboard versions for performing tasks are referred to as *key bindings*. For example, we just showed how to use the mouse to start a new script, but you can also use a key binding: Ctrl+Shift+N on Windows and command+shift+N on the Mac.

Although in this tutorial we often show how to use the mouse, **we highly recommend that you memorize key bindings for the operations you use most**. RStudio provides a useful cheat sheet with the most widely used commands. You can get it from RStudio directly:



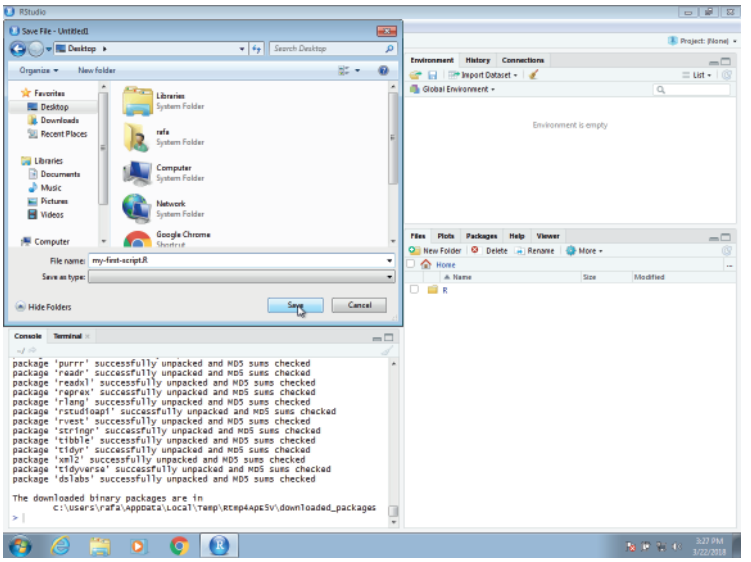
You might want to keep this handy so you can look up key bindings when you find yourself performing repetitive point-and-clicking.

1.4.3 Running commands while editing scripts

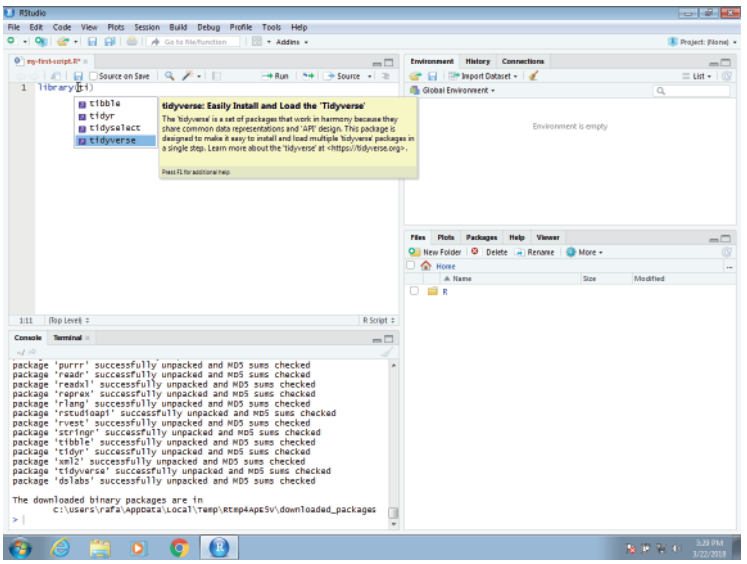
There are many editors specifically made for coding. These are useful because color and indentation are automatically added to make code more readable. RStudio is one of these editors, and it was specifically developed for R. One of the main advantages provided by RStudio over other editors is that we can test our code easily as we edit our scripts. Below we show an example.

Let's start by opening a new script as we did before. A next step is to give the script a name. We can do this through the editor by saving the current new unnamed script. To do this, click on the save icon or use the key binding Ctrl+S on Windows and command+S on the Mac.

When you ask for the document to be saved for the first time, RStudio will prompt you for a name. A good convention is to use a descriptive name, with lower case letters, no spaces, only hyphens to separate words, and then followed by the suffix `.R`. We will call this script `my-first-script.R`.



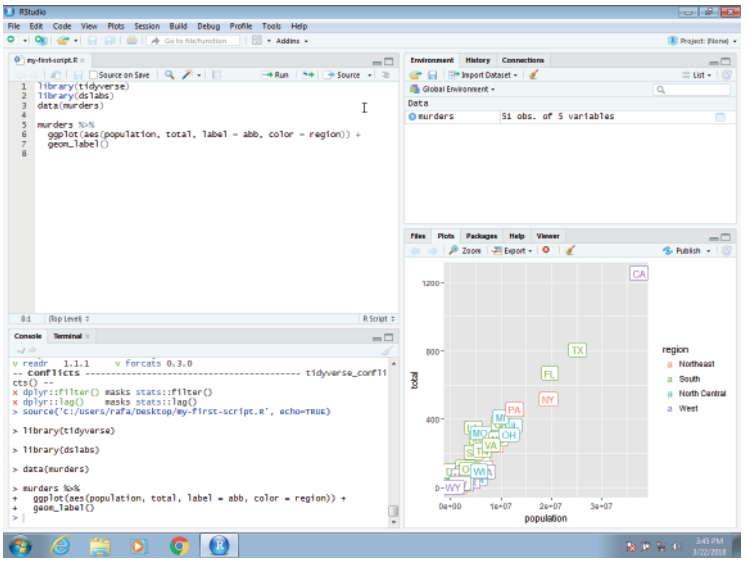
Now we are ready to start editing our first script. The first lines of code in an R script are dedicated to loading the libraries we will use. Another useful RStudio feature is that once we type `library()` it starts auto-completing with libraries that we have installed. Note what happens when we type `library(ti)`:



Another feature you may have noticed is that when you type `library(` the second parenthesis is automatically added. This will help you avoid one of the most common errors in coding: forgetting to close a parenthesis.

Now we can continue to write code. As an example, we will make a graph showing murder totals versus population totals by state. Once you are done writing the code needed to make this plot, you can try it out by *executing* the code. To do this, click on the *Run* button on the upper right side of the editing pane. You can also use the key binding: Ctrl+Shift+Enter on Windows or command+shift+return on the Mac.

Once you run the code, you will see it appear in the R console, and in this case, the generated plot appears in the plots console. Note that the plot console has a useful interface that permits you to click back and forward across different plots, zoom in to the plot, or save the plots as files.



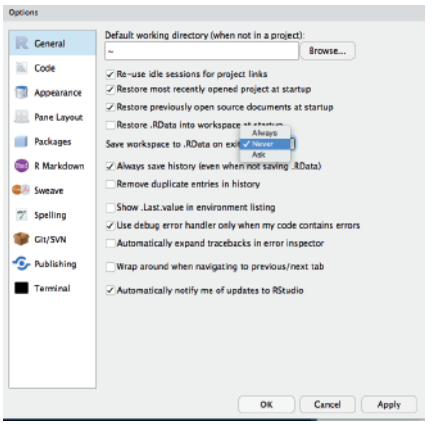
To run one line at a time instead of the entire script, you can use Control-Enter on Windows and command-return on the Mac.

1.4.4 Changing global options

You can change the look and functionality of RStudio quite a bit.

To change the global options, you click on *Tools* then *Global Options...*.

As an example, we show how to make a change that we **highly recommend**. This is to change the *Save workspace to .RData on exit* to *Never* and uncheck the *Restore .RData into workspace at start*. By default, when you exit R saves all the objects you have created into a file called .RData. This is done so that when you restart the session in the same folder, it will load these objects. We find that this causes confusion especially when we share code with colleagues and assume they have this .RData file. To change these options, make your *General* settings look like this:



1.5 Installing R packages

The functionality provided by a fresh install of R is only a small fraction of what is possible. In fact, we refer to what you get after your first install as *base R*. The extra functionality comes from add-ons available from developers. There are currently hundreds of these available from CRAN and many others shared via other repositories such as GitHub. However, because not everybody needs all available functionality, R instead makes different components available via *packages*. R makes it very easy to install packages from within R. For example, to install the **dslabs** package, which we use to share datasets and code related to this book, you would type:

```
install.packages("dslabs")
```

In RStudio, you can navigate to the *Tools* tab and select install packages. We can then load the package into our R sessions using the `library` function:

```
library(dslabs)
```

As you go through this book, you will see that we load packages without installing them. This is because once you install a package, it remains installed and only needs to be loaded with `library`. The package remains loaded until we quit the R session. If you try to load a package and get an error, it probably means you need to install it first.

We can install more than one package at once by feeding a character vector to this function:

```
install.packages(c("tidyverse", "dslabs"))
```

One advantage of using RStudio is that it auto-completes package names once you start typing, which is helpful when you do not remember the exact spelling of the package.

Note that installing **tidyverse** actually installs several packages. This commonly occurs when a package has *dependencies*, or uses functions from other packages. When you load a package using `library`, you also load its dependencies.

Once packages are installed, you can load them into R and you do not need to install them again, unless you install a fresh version of R. Remember packages are installed in R not RStudio.

It is helpful to keep a list of all the packages you need for your work in a script because if you need to perform a fresh install of R, you can re-install all your packages by simply running a script.

You can see all the packages you have installed using the following function:

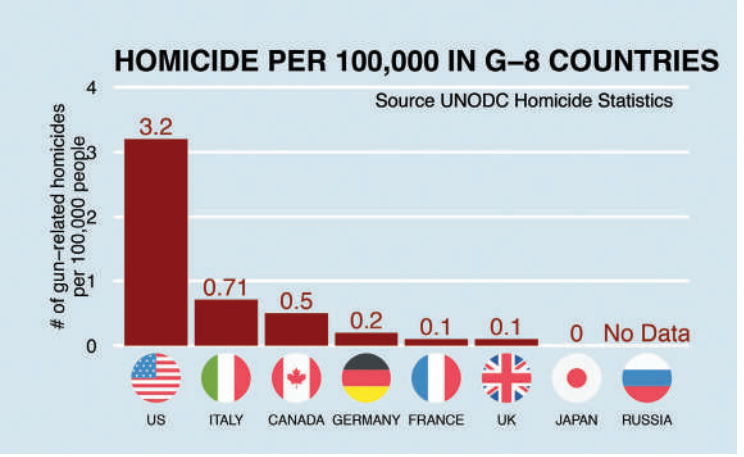
```
installed.packages()
```

2

R basics

2.1 Motivating example: US Gun Murders

Imagine you live in Europe and are offered a job in a US company with many locations across all states. It is a great job, but news with headlines such as **US Gun Homicide Rate Higher Than Other Developed Countries**[1] have you worried. Charts like this may concern you even more:

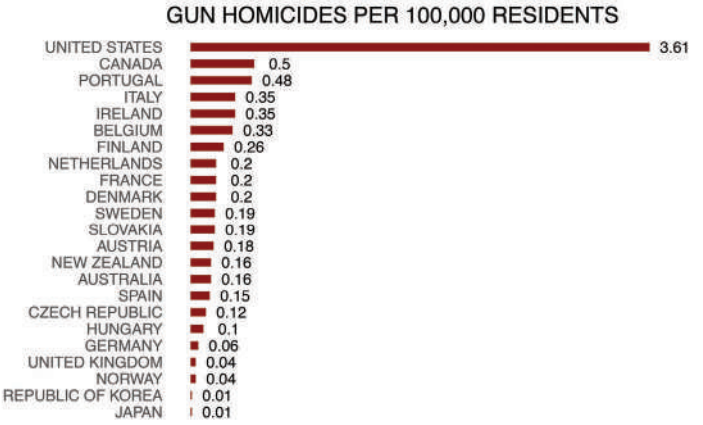


Or even worse, this version from everytown.org[2]:

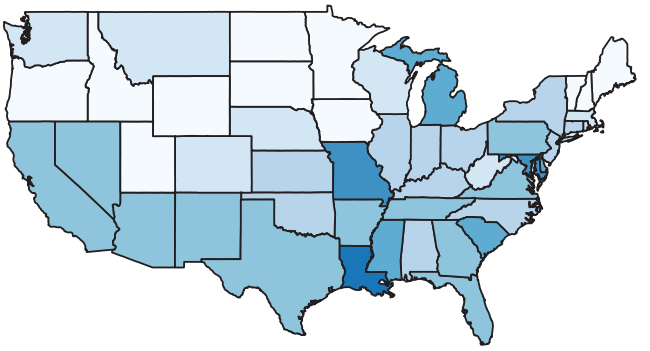
[1] http://abcnews.go.com/blogs/headlines/2012/12/us-gun-ownership-homicide-rate-higher-than-other-developed-countries/

[2] https://everytownresearch.org

DOI: 10.1201/9781003220923-2



But then you remember that the US is a large and diverse country with 50 very different states as well as the District of Columbia (DC).



California, for example, has a larger population than Canada, and 20 US states have populations larger than that of Norway. In some respects, the variability across states in the US is akin to the variability across countries in Europe. Furthermore, although not included in the charts above, the murder rates in Lithuania, Ukraine, and Russia are higher than 4 per 100,000. So perhaps the news reports that worried you are too superficial. You have options of where to live and want to determine the safety of each particular state. We will gain some insights by examining data related to gun homicides in the US during 2010 using R.

Before we get started with our example, we need to cover logistics as well as some of the very basic building blocks that are required to gain more advanced R skills. Be aware that the usefulness of some of these building blocks may not be immediately obvious, but later in the book you will appreciate having mastered these skills.

2.2 The very basics

Before we get started with the motivating dataset, we need to cover the very basics of R.

2.2.1 Objects

Suppose a high school student asks us for help solving several quadratic equations of the form $ax^2 + bx + c = 0$. The quadratic formula gives us the solutions:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

which of course change depending on the values of a, b, and c. One advantage of programming languages is that we can define variables and write expressions with these variables, similar to how we do so in math, but obtain a numeric solution. We will write out general code for the quadratic equation below, but if we are asked to solve $x^2 + x - 1 = 0$, then we define the coefficients:

```
coef_a <- 1
coef_b <- 1
coef_c <- -1
```

which stores the values for later use. We use `<-` to assign values to the variables. We can also assign values using `=` instead of `<-`, but we recommend against using `=` to avoid confusion.

Copy and paste the code above into your console to define the three variables. Note that R does not print anything when we make this assignment. This means the objects were defined successfully. Had you made a mistake, you would have received an error message.

To see the value stored in a variable, we simply ask R to evaluate `coef_a` and it shows the stored value:

```
coef_a
#> [1] 1
```

A more explicit way to ask R to show us the value stored in `coef_a` is using `print` like this:

```
print(coef_a)
#> [1] 1
```

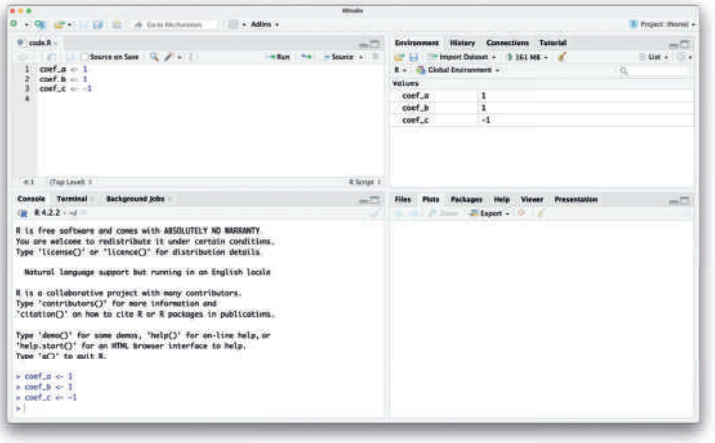
We use the term *object* to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, which are described later.

2.2.2 The workspace

As we define objects in the console, we are actually changing the *workspace*. You can see all the variables saved in your workspace by typing:

```
ls()
```

In RStudio, the *Environment* tab shows the values:



We should see `coef_a`, `coef_b`, and `coef_c`. If you try to recover the value of a variable that is not in your workspace, you receive an error. For example, if you type `x` you will receive the following message: `Error: object 'x' not found`.

Now since these values are saved in variables, to obtain a solution to our equation, we use the quadratic formula:

```
(-coef_b + sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
#> [1] 0.618
(-coef_b - sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
#> [1] -1.62
```

2.2.3 Prebuilt functions

Once you define variables, the data analysis process can usually be described as a series of *functions* applied to the data. R includes several predefined functions, and most of the analysis pipelines we construct make extensive use of these.

We already used or discussed the `install.packages`, `library`, and `ls` functions. We also used the function `sqrt` to solve the quadratic equation above. There are many more prebuilt functions, and even more can be added through packages. These functions do not appear in the workspace because you did not define them, but they are available for immediate use.

In general, we need to use parentheses to evaluate a function. If you type `ls`, the function is not evaluated and instead R shows you the code that defines the function. If you type `ls()` the function is evaluated and, as seen above, we see objects in the workspace.

Unlike `ls`, most functions require one or more *arguments*. Below is an example of how we assign an object to the argument of the function `log`. Remember that we earlier defined `coef_a` to be 1:

```
log(8)
#> [1] 2.08
log(coef_a)
```

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

You can find out what the function expects and what it does by reviewing the very useful manuals included in R. You can get help by using the `help` function like this:

```
help("log")
```

For most functions, we can also use this shorthand:

```
?log
```

The help page will show you what arguments the function is expecting. For example, a `log` needs `x` and `base` to run. However, some arguments are required and others are optional. You can determine which arguments are optional by noting in the help document that a default value is assigned with `=`. Defining these is optional. For example, the base of the function `log` defaults to `base = exp(1)` making `log` the natural log by default.

If you want a quick look at the arguments without opening the help system, you can type:

```
args(log)
#> function (x, base = exp(1))
#> NULL
```

You can change the default values by simply assigning another object:

```
log(8, base = 2)
#> [1] 3
```

Note that we have not been specifying the argument `x` as such:

```
log(x = 8, base = 2)
#> [1] 3
```

The above code works, but we can save ourselves some typing: if no argument name is used, R assumes you are entering arguments in the order shown in the help file or by `args`. So by not using the names, it assumes the arguments are `x` followed by `base`:

```
log(8, 2)
#> [1] 3
```

If using the arguments' names, then we can include them in whatever order we want:

```
log(base = 2, x = 8)
#> [1] 3
```

To specify arguments, we must use `=`, and cannot use `<-`.

There are some exceptions to the rule that functions need the parentheses to be evaluated. Among these, the most commonly used are the arithmetic and relational operators. For example:

```
2^3
#> [1] 8
```

You can see the arithmetic operators by typing:

```
help("+")
```

or

```
?"+"
```

and the relational operators by typing:

```
help(">")
```

or

```
?">"
```

2.2.4 Prebuilt objects

There are several datasets that are included for users to practice and test out functions. You can see all the available datasets by typing:

```
data()
```

This shows you the object name for these datasets. These datasets are objects that can be used by simply typing the name. For example, if you type:

```
co2
```

R will show you Mauna Loa atmospheric CO2 concentration data.

Other prebuilt objects are mathematical quantities, such as the constant π and ∞:

```
pi
#> [1] 3.14
Inf + 1
#> [1] Inf
```

2.2.5 Variable names

We used `coef_a`, `coef_b`, and `coef_c` as variable names, but variable names can be almost anything. When writing code in R, it's important to choose variable names that are both meaningful and avoid conflicts with existing functions or reserved words in the language. For example, we did not use `a`, `b`, and `c` to avoid a conflict with the `c()` function in R, described in Section 2.4.1. If you were to name a variable `c`, you would not receive an error or warning, but the conflict can lead to unexpected behavior and bugs that are hard to diagnose.

Some basic rules in R are that variable names have to start with a letter, can't contain spaces, and should not be variables that are predefined in R, such as `c`.

A nice convention to follow is to use meaningful words that describe what is stored, use only lowercase, and use underscores as a substitute for spaces. For the quadratic equations, we could use something like this for the two roots:

```
r_1 <- (-coef_b + sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
r_2 <- (-coef_b - sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
```

For more advice, we highly recommend studying Hadley Wickham's style guide[3].

2.2.6 Saving your workspace

Values remain in the workspace until you end your session or erase them with the function `rm`. But workspaces also can be saved for later use. In fact, when you quit R, the program asks you if you want to save your workspace. If you do save it, the next time you start R, the program will restore the workspace.

We actually recommend against saving the workspace this way because, as you start working on different projects, it will become harder to keep track of what is saved. Instead, we recommend you assign the workspace a specific name. You can do this by using the function `save` or `save.image`. To load, use the function `load`. When saving a workspace, we recommend the suffix `rda` or `RData`. In RStudio, you can also do this by navigating to the *Session* tab and choosing *Save Workspace as*. You can later load it using the *Load Workspace* options in the same tab. You can read the help pages on `save`, `save.image`, and `load` to learn more.

2.2.7 Why use scripts?

To solve another equation such as $3x^2 + 2x - 1$, we can copy and paste the code above and then redefine the variables and recompute the solution:

```
coef_a <- 3
coef_b <- 2
coef_c <- -1
(-coef_b + sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
(-coef_b - sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
```

By creating and saving a script with the code above, we would not need to retype everything each time and, instead, simply change the variable values. Try writing the script above into an editor and notice how easy it is to change the variables and receive an answer.

2.2.8 Commenting your code

If a line of R code starts with the symbol `#`, it is a comment and is not evaluated. We can use this to write reminders of why we wrote particular code. For example, in the script above, we could add:

[3] http://adv-r.had.co.nz/Style.html

```
## Code to compute solution to quadratic equation

## Define the variables
coef_a <- 3
coef_b <- 2
coef_c <- -1

## Now compute the solution
(-coef_b + sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
(-coef_b - sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a)
```

> **Note**
>
> You are ready to do exercises 1–5.

2.3 Data types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function `class` helps us determine what type of object we have:

```
a <- 2
class(a)
#> [1] "numeric"
```

To work efficiently in R, it is important to learn the different types of variables and what we can do with these.

2.3.1 Data frames

Up to now, the variables we have defined are just one number. This is not very useful for storing data. The most common way of storing a dataset in R is in a *data frame*. Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns. Data frames are particularly useful for datasets because we can combine different data types into one object.

A large proportion of data analysis challenges start with data stored in a data frame. For example, we stored the data for our motivating example in a data frame. You can access this dataset by loading the **dslabs** library, which provides the `murders` dataset:

```
library(dslabs)
```

To see that this is in fact a data frame, we type:

```
class(murders)
#> [1] "data.frame"
```

2.3.2 Examining objects

The function `str` is useful for finding out more about the structure of an object:

```
str(murders)
#> 'data.frame':    51 obs. of  5 variables:
#> $ state : chr "Alabama" "Alaska" "Arizona" "Arkansas" ...
#> $ abb : chr "AL" "AK" "AZ" "AR" ...
#> $ region : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2
#>    2 ...
#> $ population: num 4779736 710231 6392017 2915918 37253956 ...
#> $ total : num 135 19 232 93 1257 ...
```

This tells us much more about the object. We see that the table has 51 rows (50 states plus DC) and five variables. We can show the first six lines using the function `head`:

```
head(murders)
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

In this dataset, each state is considered an observation and five variables are reported for each state.

Before we go any further in answering our original question about different states, let's learn more about the components of this object.

2.3.3 The accessor: $

For our analysis, we will need to access the different variables represented by columns included in this data frame. To do this, we use the accessor operator `$` in the following way:

```
murders$population
#>  [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097
#>  [8]   897934   601723 19687653  9920000  1360301  1567582 12830632
#> [15]  6483802  3046355  2853118  4339367  4533372  1328361  5773552
#> [22]  6547629  9883640  5303925  2967297  5988927   989415  1826341
#> [29]  2700551  1316470  8791894  2059179 19378102  9535483   672591
#> [36] 11536504  3751351  3831074 12702379  1052567  4625364   814180
#> [43]  6346105 25145561  2763885   625741  8001024  6724540  1852994
#> [50]  5686986   563626
```

But how did we know to use `population`? Previously, by applying the function `str` to the object `murders`, we revealed the names for each of the five variables stored in this table. We can quickly access the variable names using:

```
names(murders)
#> [1] "state"      "abb"        "region"     "population" "total"
```

It is important to know that the order of the entries in `murders$population` preserves the order of the rows in our data table. This will later permit us to manipulate one variable based on the results of another. For example, we will be able to order the state names by the number of murders.

> **Tip**
>
> R comes with a very nice auto-complete functionality that saves us the trouble of typing out all the names. Try typing `murders$p` then hitting the *tab* key on your keyboard. This functionality and many other useful auto-complete features are available when working in RStudio.

2.3.4 Vectors

The object `murders$population` is not one number but several. We call these types of objects *vectors*. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function `length` tells you how many entries are in the vector:

```
pop <- murders$population
length(pop)
#> [1] 51
```

This particular vector is *numeric* since population sizes are numbers:

```
class(pop)
#> [1] "numeric"
```

In a numeric vector, every entry must be a number.

To store character strings, vectors can also be of class *character*. For example, the state names are characters:

```
class(murders$state)
#> [1] "character"
```

As with numeric vectors, all entries in a character vector need to be a character.

Another important type of vectors are *logical vectors*. These must be either `TRUE` or `FALSE`.

```
z <- 3 == 2
z
#> [1] FALSE
class(z)
#> [1] "logical"
```

Here the `==` is a relational operator asking if 3 is equal to 2. In R, if you just use one `=`, you actually assign a variable, but if you use two `==` you test for equality.

You can see the other *relational operators* by typing:

```
?Comparison
```

In future sections, you will see how useful relational operators can be.

We discuss more important features of vectors after the next set of exercises.

> **Note**
>
> Mathematically, the values in `pop` are integers and there is an integer class in R. However, by default, numbers are assigned class numeric even when they are round integers. For example, `class(1)` returns numeric. You can turn them into class integer with the `as.integer()` function or by adding an `L` like this: `1L`. Note the class by typing: `class(1L)`

2.3.5 Factors

In the `murders` dataset, we might expect the region to also be a character vector. However, it is not:

```
class(murders$region)
#> [1] "factor"
```

It is a *factor*. Factors are useful for storing categorical data. We can see that there are only 4 regions by using the `levels` function:

```
levels(murders$region)
#> [1] "Northeast"     "South"         "North Central" "West"
```

In the background, R stores these *levels* as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

Note that the levels have an order that is different from the order of appearance in the factor object. The default in R is for the levels to follow alphabetical order. However, often we want the levels to follow a different order. You can specify an order through the `levels` argument when creating the factor with the `factor` function. For example, in the murders dataset regions are ordered from east to west. The function `reorder` lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector. We will demonstrate this with a simple example, and will see more advanced ones in the Data Visualization part of the book.

Suppose we want the levels of region ordered by the total number of murders rather than alphabetically. If there are values associated with each level, we can use the `reorder` and specify a data summary to determine the order. The following code takes the sum of the total murders in each region, and reorders the factor following these sums.

```
region <- murders$region
value <- murders$total
region <- reorder(region, value, FUN = sum)
levels(region)
#> [1] "Northeast"     "North Central" "West"          "South"
```

The new order is in agreement with the fact that the Northeast has the least murders and the South has the most.

> **Warning**
>
> Factors can be a source of confusion since sometimes they behave like characters and sometimes they do not. As a result, confusing factors and characters are a common source of bugs.

2.3.6 Lists

Data frames are a special case of *lists*. Lists are useful because you can store any combination of different types. You can create a list using the `list` function like this:

```
record <- list(name = "John Doe",
               student_id = 1234,
               grades = c(95, 82, 91, 97, 93),
               final_grade = "A")
```

The function `c` is described in Section 2.4.

This list includes a character, a number, a vector with five numbers, and another character.

```
record
#> $name
#> [1] "John Doe"
#>
#> $student_id
#> [1] 1234
#>
#> $grades
#> [1] 95 82 91 97 93
#>
#> $final_grade
#> [1] "A"
class(record)
#> [1] "list"
```

As with data frames, you can extract the components of a list with the accessor `$`.

```
record$student_id
#> [1] 1234
```

We can also use double square brackets (`[[`) like this:

```
record[["student_id"]]
#> [1] 1234
```

You should get used to the fact that in R, there are often several ways to do the same thing, such as accessing entries.

You might also encounter lists without variable names.

```
record2 <- list("John Doe", 1234)
record2
#> [[1]]
#> [1] "John Doe"
#>
#> [[2]]
#> [1] 1234
```

If a list does not have names, you cannot extract the elements with `$`, but you can still use the brackets method and instead of providing the variable name, you provide the list index, like this:

```
record2[[1]]
#> [1] "John Doe"
```

We won't be using lists until later, but you might encounter one in your own exploration of R. For this reason, we show you some basics here.

2.3.7 Matrices

Matrices are another type of object that are common in R. Matrices are similar to data frames in that they are two-dimensional: they have rows and columns. However, like numeric, character and logical vectors, entries in matrices have to be all the same type. For this reason data frames are much more useful for storing data, since we can have characters, factors, and numbers in them.

Yet matrices have a major advantage over data frames: we can perform matrix algebra operations, a powerful type of mathematical technique. We do not describe these operations in this book, but much of what happens in the background when you perform a data analysis involves matrices. We only cover matrices briefly here since some of the functions we will learn return matrices. However, if you plan to perform more advanced work, we highly recommend learning more as they are widely used in data analysis.

We can define a matrix using the `matrix` function. We need to specify the data in the matrix as well as the number of rows and columns.

```
mat <- matrix(1:12, 4, 3)
mat
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

The shorthand using `:` is described in Section 2.4.

You can access specific entries in a matrix using square brackets (`[`). If you want the second row, third column, you use:

```
mat[2, 3]
#> [1] 10
```

If you want the entire second row, you leave the column spot empty:

```
mat[2, ]
#> [1]  2  6 10
```

Notice that this returns a vector, not a matrix.

Similarly, if you want the entire third column, you leave the row spot empty:

```
mat[, 3]
#> [1]  9 10 11 12
```

This is also a vector, not a matrix.

You can access more than one column or more than one row if you like. This will give you a new matrix.

```
mat[, 2:3]
#>      [,1] [,2]
#> [1,]    5    9
#> [2,]    6   10
#> [3,]    7   11
#> [4,]    8   12
```

You can subset both rows and columns:

```
mat[1:2, 2:3]
#>      [,1] [,2]
#> [1,]    5    9
#> [2,]    6   10
```

We can convert matrices into data frames using the function `as.data.frame`:

```
as.data.frame(mat)
#>   V1 V2 V3
#> 1  1  5  9
#> 2  2  6 10
#> 3  3  7 11
#> 4  4  8 12
```

You can also use single square brackets (`[`) to access rows and columns of a data frame:

```
murders[25, 1]
#> [1] "Mississippi"
murders[2:3, ]
#>     state abb region population total
#> 2  Alaska  AK   West     710231    19
#> 3 Arizona  AZ   West    6392017   232
```

> **Note**
>
> You are ready to do exercises 6–11.

2.4 Vectors

In R, the most basic objects available to store data are *vectors*. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

2.4.1 Creating vectors

We can create vectors using the function `c`, which stands for *concatenate*. We use `c` to concatenate entries in the following way:

```
codes <- c(380, 124, 818)
codes
#> [1] 380 124 818
```

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variable names.

```
country <- c("italy", "canada", "egypt")
```

In R you can also use single quotes:

```
country <- c('italy', 'canada', 'egypt')
```

But be careful not to confuse the single quote ' with the *back quote* `.

By now you should know that if you type:

```
country <- c(italy, canada, egypt)
```

you receive an error because the variables `italy`, `canada`, and `egypt` are not defined. If we do not use the quotes, R looks for variables with those names and returns an error.

2.4.2 Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes, we can use the names to connect the two:

```
codes <- c(italy = 380, canada = 124, egypt = 818)
codes
#>  italy canada  egypt
#>    380    124    818
```

The object `codes` continues to be a numeric vector:

```
class(codes)
#> [1] "numeric"
```

but with names:

```
names(codes)
#> [1] "italy"  "canada" "egypt"
```

If the use of strings without quotes looks confusing, know that you can use the quotes as well:

```
codes <- c("italy" = 380, "canada" = 124, "egypt" = 818)
codes
#>  italy canada  egypt
#>    380    124    818
```

There is no difference between this function call and the previous one. This is one of the many ways in which R is quirky compared to other languages.

We can also assign names using the `names` functions:

```
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
#>  italy canada  egypt
#>    380    124    818
```

2.4.3 Sequences

Another useful function for creating vectors generates sequences:

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

The first argument defines the start, and the second defines the end, which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump by:

```
seq(1, 10, 2)
#> [1] 1 3 5 7 9
```

If we want consecutive integers, we can use the following shorthand:

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

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
#> [1] "integer"
```

However, if we create a sequence including non-integers, the class changes:

```
class(seq(1, 10, 0.5))
#> [1] "numeric"
```

2.4.4 Subsetting

We use square brackets to access specific elements of a vector. For the vector `codes` we defined above, we can access the second element using:

```
codes[2]
#> canada
#>    124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
#> italy egypt
#>   380   818
```

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```
codes[1:2]
#>  italy canada
#>    380    124
```

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
#> canada
#>    124
codes[c("egypt","italy")]
#> egypt italy
#>   818   380
```

2.5 Coercion

In general, *coercion* is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion. Failing to understand *coercion* can drive programmers crazy when attempting to code in R since it behaves quite differently from most other languages in this regard. Let's learn about it with some examples.

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at `x` and its class:

```
x
#> [1] "1"      "canada" "3"
class(x)
#> [1] "character"
```

R *coerced* the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings `"1"` and `"3"`. The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to change from one type to another. For example, you can turn numbers into characters with:

```
x <- 1:5
y <- as.character(x)
y
#> [1] "1" "2" "3" "4" "5"
```

You can turn it back with `as.numeric`:

```
as.numeric(y)
#> [1] 1 2 3 4 5
```

This function is actually quite useful since datasets that include numbers as character strings are common.

2.6 Not availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an `NA` for "not available." For example:

```
x <- c("1", "b", "3")
as.numeric(x)
#> Warning: NAs introduced by coercion
#> [1]  1 NA  3
```

R does not have any guesses for what number you want when you type `b`, so it does not try.

As a data scientist you will encounter the `NA`s often as they are generally used for missing data, a common problem in real-world datasets.

> **Note**
>
> You are ready to do exercises 12–23.

2.7 Sorting

Now that we have mastered some basic R knowledge, let's try to gain some insights into the safety of different states in the context of gun murders.

2.7.1 `sort`

Say we want to rank the states from least to most gun murders. The function `sort` sorts a vector in increasing order. We can therefore see the largest number of gun murders by typing:

```
library(dslabs)
sort(murders$total)
#>  [1]    2    4    5    5    7    8   11   12   12   16   19   21   22
#> [14]   27   32   36   38   53   63   65   67   84   93   93   97   97
#> [27]   99  111  116  118  120  135  142  207  219  232  246  250  286
#> [40]  293  310  321  351  364  376  413  457  517  669  805 1257
```

However, this does not give us information about which states have which murder totals. For example, we don't know which state had 1257.

2.7.2 `order`

The function `order` is closer to what we want. It takes a vector as input and returns the vector of indexes that sorts the input vector. This may sound confusing so let's look at a simple example. We can create a vector and sort it:

```
x <- c(31, 4, 15, 92, 65)
sort(x)
#> [1]  4 15 31 65 92
```

Rather than sort the input vector, the function `order` returns the index that sorts input vector:

```
index <- order(x)
x[index]
#> [1]  4 15 31 65 92
```

This is the same output as that returned by `sort(x)`. If we look at this index, we see why it works:

```
x
#> [1] 31  4 15 92 65
order(x)
#> [1] 2 3 1 5 4
```

The second entry of `x` is the smallest, so `order(x)` starts with `2`. The next smallest is the third entry, so the second entry is `3`, etc.

How does this help us order the states by murders? First, remember that the entries of vectors you access with `$` follow the same order as the rows in the table. For example, these two vectors containing state names and abbreviations, respectively, are matched by their order:

```
murders$state[1:6]
#> [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
#> [6] "Colorado"
murders$abb[1:6]
#> [1] "AL" "AK" "AZ" "AR" "CA" "CO"
```

This means we can order the state names by their total murders. We first obtain the index that orders the vectors according to murder totals and then index the state names vector:

```
ind <- order(murders$total)
murders$abb[ind]
#>  [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT" "RI" "AK" "IA" "UT"
#> [14] "WV" "NE" "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI"
#> [27] "DC" "OK" "KY" "MA" "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC"
#> [40] "MD" "OH" "MO" "LA" "IL" "GA" "MI" "PA" "NY" "FL" "TX" "CA"
```

According to the above, California had the most murders.

2.7.3 `max` and `which.max`

If we are only interested in the entry with the largest value, we can use `max` for the value:

```
max(murders$total)
#> [1] 1257
```

and `which.max` for the index of the largest value:

```
i_max <- which.max(murders$total)
murders$state[i_max]
#> [1] "California"
```

For the minimum, we can use `min` and `which.min` in the same way.

Does this mean California is the most dangerous state? In an upcoming section, we argue that we should be considering rates instead of totals. Before doing that, we introduce one last order-related function: `rank`.

2.7.4 `rank`

Although not as frequently used as `order` and `sort`, the function `rank` is also related to order and can be useful. For any given vector it returns a vector with the rank of the first entry, second entry, etc., of the input vector. Here is a simple example:

```
x <- c(31, 4, 15, 92, 65)
rank(x)
#> [1] 3 1 2 5 4
```

To summarize, let's look at the results of the three functions we have introduced:

original	sort	order	rank
31	4	2	3
4	15	3	1
15	31	1	2
92	65	5	5
65	92	4	4

> **Note**
>
> You are ready to do exercises 24–31

2.8 Vector arithmetics

California had the most murders, but does this mean it is the most dangerous state? What if it just has many more people than any other state? We can quickly confirm that California indeed has the largest population:

```
library(dslabs)
murders$state[which.max(murders$population)]
#> [1] "California"
```

with over 37 million inhabitants. It is therefore unfair to compare the totals if we are interested in learning how safe the state is. What we really should be computing is the murders per capita. The reports we describe in the motivating section used murders per 100,000 as the unit. To compute this quantity, the powerful vector arithmetic capabilities of R come in handy.

2.8.1 Rescaling a vector

In R, arithmetic operations on vectors occur *element-wise*. For a quick example, suppose we have height in inches:

```
inches <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
```

and want to convert to centimeters. Notice what happens when we multiply `inches` by 2.54:

```
inches * 2.54
#>  [1] 175 157 168 178 178 185 170 185 170 178
```

In the line above, we multiplied each element by 2.54. Similarly, if for each entry we want to compute how many inches taller or shorter than 69 inches, the average height for males, we can subtract it from every entry like this:

```
inches - 69
#>  [1]  0 -7 -3  1  1  4 -2  4 -2  1
```

2.8.2 Two vectors

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a+e \\ b+f \\ c+g \\ d+h \end{pmatrix}$$

The same holds for other mathematical operations, such as `-`, `*`, and `/`.

This implies that to compute the murder rates, we can simply type:

```
murder_rate <- murders$total / murders$population * 100000
```

Once we do this, we notice that California is no longer near the top of the list. In fact, we can use what we have learned to order the states by murder rate:

```
murders$abb[order(murder_rate)]
#>  [1] "VT" "NH" "HI" "ND" "IA" "ID" "UT" "ME" "WY" "OR" "SD" "MN" "MT"
#> [14] "CO" "WA" "WV" "RI" "WI" "NE" "MA" "IN" "KS" "NY" "KY" "AK" "OH"
#> [27] "CT" "NJ" "AL" "IL" "OK" "NC" "NV" "VA" "AR" "TX" "NM" "CA" "FL"
#> [40] "TN" "PA" "AZ" "GA" "MS" "MI" "DE" "SC" "MD" "MO" "LA" "DC"
```

2.8.3 Beware of recycling

Another common source of unnoticed errors in R is the use of *recycling*. We saw that vectors are added elementwise. So if the vectors don't match in length, it is natural to assume that we should get an error. But we don't. Notice what happens:

```
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50, 60, 70)
x + y
#> Warning in x + y: longer object length is not a multiple of shorter
#> object length
#> [1] 11 22 33 41 52 63 71
```

We do get a warning, but no error. For the output, R has recycled the numbers in `x`. Notice the last digit of numbers in the output.

> **Note**
>
> You are now ready to do exercises 32–34.

2.9 Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector. In this section, we continue working with our US murders example, which we can load like this:

```
library(dslabs)
```

2.9.1 Subsetting with logicals

We have now calculated the murder rate using:

```
murder_rate <- murders$total / murders$population * 100000
```

Imagine you are moving from Italy where, according to an ABC news report, the murder rate is only 0.71 per 100,000. You would prefer to move to a state with a similar murder rate. Another powerful feature of R is that we can use logicals to index vectors. If we compare a vector to a single number, it actually performs the test for each entry. The following is an example related to the question above:

```
ind <- murder_rate < 0.71
```

If we instead want to know if a value is less or equal, we can use:

```
ind <- murder_rate <= 0.71
```

Note that we get back a logical vector with `TRUE` for each entry smaller than or equal to 0.71. To see which states these are, we can leverage the fact that vectors can be indexed with logicals.

```
murders$state[ind]
#> [1] "Hawaii"        "Iowa"          "New Hampshire" "North Dakota"
#> [5] "Vermont"
```

In order to count how many are TRUE, the function `sum` returns the sum of the entries of a vector and logical vectors get *coerced* to numeric with `TRUE` coded as 1 and `FALSE` as 0. Thus we can count the states using:

```
sum(ind)
#> [1] 5
```

2.9.2 Logical operators

Suppose we like the mountains and we want to move to a safe state in the western region of the country. We want the murder rate to be at most 1. In this case, we want two different things to be true. Here we can use the logical operator *and*, which in R is represented with `&`. This operation results in `TRUE` only when both logicals are `TRUE`. To see this, consider this example:

```
TRUE & TRUE
#> [1] TRUE
TRUE & FALSE
#> [1] FALSE
FALSE & FALSE
#> [1] FALSE
```

For our example, we can form two logicals:

```
west <- murders$region == "West"
safe <- murder_rate <= 1
```

and we can use the `&` to get a vector of logicals that tells us which states satisfy both conditions:

```
ind <- safe & west
murders$state[ind]
#> [1] "Hawaii"  "Idaho"   "Oregon"  "Utah"    "Wyoming"
```

2.9.3 `which`

Suppose we want to look up California's murder rate. For this type of operation, it is convenient to convert vectors of logicals into indexes instead of keeping long vectors of logicals. The function `which` tells us which entries of a logical vector are TRUE. So we can type:

```
ind <- which(murders$state == "California")
murder_rate[ind]
#> [1] 3.37
```

2.9.4 `match`

If instead of just one state we want to find out the murder rates for several states, say New York, Florida, and Texas, we can use the function `match`. This function tells us which indexes of a second vector match each of the entries of a first vector:

```
ind <- match(c("New York", "Florida", "Texas"), murders$state)
ind
#> [1] 33 10 44
```

Now we can look at the murder rates:

```
murder_rate[ind]
#> [1] 2.67 3.40 3.20
```

2.9.5 `%in%`

If rather than an index we want a logical that tells us whether or not each element of a first vector is in a second, we can use the function `%in%`. Let's imagine you are not sure if Boston, Dakota, and Washington are states. You can find out like this:

```
c("Boston", "Dakota", "Washington") %in% murders$state
#> [1] FALSE FALSE  TRUE
```

Note that we will be using `%in%` often throughout the book.

There is a connection between `match` and `%in%` through `which`. To see this, notice that the following two lines produce the same index (although in different order):

```
match(c("New York", "Florida", "Texas"), murders$state)
#> [1] 33 10 44
which(murders$state %in% c("New York", "Florida", "Texas"))
#> [1] 10 33 44
```

> **Note**
>
> You are now ready to do exercises 35–42.

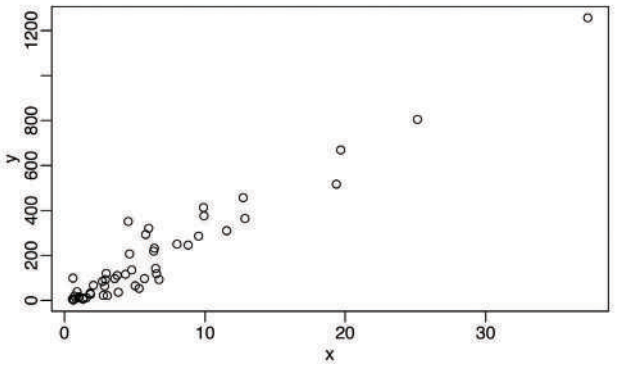
2.10 Basic plots

In Chapter 8 we describe an add-on package that provides a powerful approach to producing plots in R. We then have an entire part on Data Visualization in which we provide many examples. Here we briefly describe some of the functions that are available in a basic R installation.

2.10.1 `plot`

The `plot` function can be used to make scatterplots. Here is a plot of total murders versus population.

```
x <- murders$population / 10^6
y <- murders$total
plot(x, y)
```



For a quick plot that avoids accessing variables twice, we can use the `with` function:

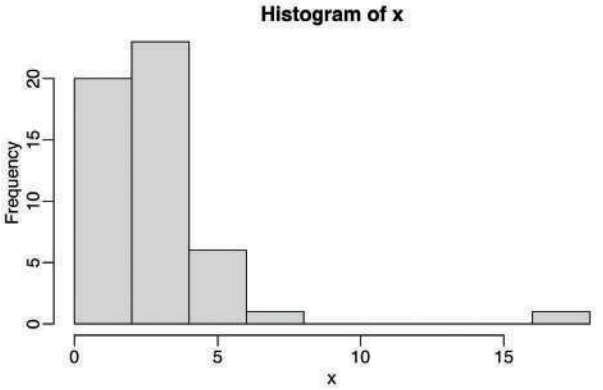
```
with(murders, plot(population, total))
```

The function `with` lets us use the `murders` column names in the `plot` function. It also works with any data frames and any function.

2.10.2 `hist`

We will describe histograms as they relate to distributions in the Data Visualization part of the book. Here we will simply note that histograms are a powerful graphical summary of a list of numbers that gives you a general overview of the types of values you have. We can make a histogram of our murder rates by simply typing:

```
x <- with(murders, total / population * 100000)
hist(x)
```



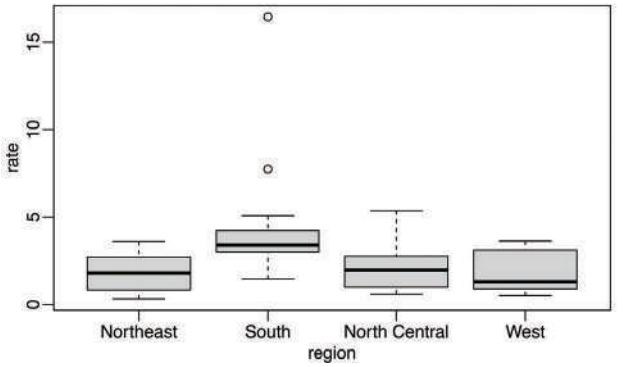
We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

```
murders$state[which.max(x)]
#> [1] "District of Columbia"
```

2.10.3 `boxplot`

Boxplots will also be described in the Data Visualization part of the book. They provide a more terse summary than histograms, but they are easier to stack with other boxplots. For example, here we can use them to compare the different regions:

```
murders$rate <- with(murders, total / population * 100000)
boxplot(rate~region, data = murders)
```

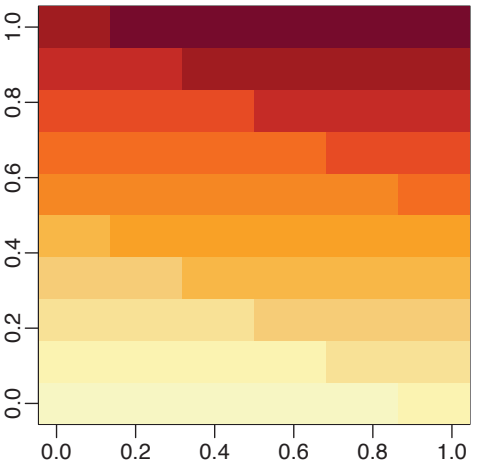


We can see that the South has higher murder rates than the other three regions.

2.10.4 `image`

The image function displays the values in a matrix using color. Here is a quick example:

```
x <- matrix(1:120, 12, 10)
image(x)
```



> **Note**
>
> You are now ready to do exercises 43–45.

2.11 Exercises

1. What is the sum of the first 100 positive integers? The formula for the sum of integers 1 through n is $n(n+1)/2$. Define $n = 100$ and then use R to compute the sum of 1 through 100 using the formula. What is the sum?
2. Now use the same formula to compute the sum of the integers from 1 to 1000.
3. Look at the result of typing the following code into R:

   ```
   n <- 1000
   x <- seq(1, n)
   sum(x)
   ```

   Based on the result, what do you think the functions `seq` and `sum` do? You can use `help`.

   (a) `sum` creates a list of numbers and `seq` adds them up.
   (b) `seq` creates a list of numbers and `sum` adds them up.
   (c) `seq` creates a random list and `sum` computes the sum of 1 through 1,000.
   (d) `sum` always returns the same number.

4. In math and programming, we say that we evaluate a function when we replace the argument with a given value. So if we type `sqrt(4)`, we evaluate the `sqrt` function. In R, you can evaluate a function inside another function. The evaluations happen from the inside out. Use one line of code to compute the log, in base 10, of the square root of 100.

5. Which of the following will always return the numeric value stored in `x`? You can try out examples and use the help system if you want.

   (a) `log(10^x)`
   (b) `log10(x^10)`
   (c) `log(exp(x))`
   (d) `exp(log(x, base = 2))`

6. Make sure the US murders dataset is loaded. Use the function `str` to examine the structure of the `murders` object. Which of the following best describes the variables represented in this data frame?

   (a) The 51 states.
   (b) The murder rates for all 50 states and DC.
   (c) The state name, the abbreviation of the state name, the state's region, and the state's population and total number of murders for 2010.
   (d) `str` shows no relevant information.

7. What are the column names used by the data frame for these five variables?

8. Use the accessor `$` to extract the state abbreviations and assign them to the object `a`. What is the class of this object?

9. Now use the square brackets to extract the state abbreviations and assign them to the object `b`. Use the `identical` function to determine if `a` and `b` are the same.

10. We saw that the `region` column stores a factor. You can corroborate this by typing:

    ```
    class(murders$region)
    ```

    With one line of code, use the functions `levels` and `length` to determine the number of regions defined by this dataset.

11. The function `table` takes a vector and returns the frequency of each element. You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of number of states per region.

12. Use the function `c` to create a vector with the average high temperatures in January for Beijing, Lagos, Paris, Rio de Janeiro, San Juan, and Toronto, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object `temp`.

13. Now create a vector with the city names and call the object `city`.

14. Use the `names` function and the objects defined in the previous exercises to associate the temperature data with its corresponding city.

15. Use the `[` and `:` operators to access the temperature of the first three cities on the list.

16. Use the `[` operator to access the temperature of Paris and San Juan.

17. Use the `:` operator to create a sequence of numbers $12, 13, 14, \ldots, 73$.

18. Create a vector containing all the positive odd numbers smaller than 100.

19. Create a vector of numbers that starts at 6, does not pass 55, and adds numbers in increments of 4/7: 6, 6 + 4/7, 6 + 8/7, etc. How many numbers does the list have? Hint: use `seq` and `length`.

20. What is the class of the following object `a <- seq(1, 10, 0.5)`?

21. What is the class of the following object `a <- seq(1, 10)`?

22. The class of `class(a<-1)` is numeric, not integer. R defaults to numeric and to force an integer, you need to add the letter `L`. Confirm that the class of `1L` is integer.

23. Define the following vector:

```
x <- c("1", "3", "5")
```

and coerce it to get integers.

24. For exercises 24–31, we will use the US murders dataset. Make sure you load it prior to starting. Use the `$` operator to access the population size data and store it as the object `pop`. Then use the `sort` function to redefine `pop` so that it is sorted. Finally, use the `[` operator to report the smallest population size.

25. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use `order` instead of `sort`.

26. We can actually perform the same operation as in the previous exercise using the function `which.min`. Write one line of code that does this.

27. Now we know how small the smallest state is and we know which row represents it. Which state is it? Define a variable `states` to be the state names from the `murders` data frame. Report the name of the state with the smallest population.

28. You can create a data frame using the `data.frame` function. Here is a quick example:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature = temp)
```

Use the `rank` function to determine the population rank of each state from smallest population size to biggest. Save these ranks in an object called `ranks`, then create a data frame with the state name and its rank. Call the data frame `my_df`.

29. Repeat the previous exercise, but this time order `my_df` so that the states are ordered from least populous to most populous. Hint: create an object `ind` that stores the indexes needed to order the population values. Then use the bracket operator `[` to re-order each column in the data frame.

30. The `na_example` vector represents a series of counts. You can quickly examine the object using:

```
str(na_example)
#>  int [1:1000] 2 1 3 2 1 3 1 4 3 2 ...
```

However, when we compute the average with the function `mean`, we obtain an `NA`:

```
mean(na_example)
#> [1] NA
```

The `is.na` function returns a logical vector that tells us which entries are `NA`. Assign this logical vector to an object called `ind` and determine how many `NA`s does `na_example` have.

31. Now compute the average again, but only for the entries that are not `NA`. Hint: remember the `!` operator, which turns `FALSE` into `TRUE` and vice versa.

32. In exercises 28 we created the `temp` data frame:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature = temp)
```

Remake the data frame using the code above, but add a line that converts the temperature from Fahrenheit to Celsius. The conversion is $C = \frac{5}{9} \times (F - 32)$.

33. What is the following sum $1 + 1/2^2 + 1/3^2 + \cdots + 1/100^2$? Hint: thanks to Euler, we know it should be close to $\pi^2/6$.

34. Compute the per 100,000 murder rate for each state and store it in the object `murder_rate`. Then compute the average murder rate for the US using the function `mean`. What is the average?

35. For remaining exercises 35–42, start by loading the library and data.

```
library(dslabs)
```

Compute the per 100,000 murder rate for each state and store it in an object called `murder_rate`. Then use logical operators to create a logical vector named `low` that tells us which entries of `murder_rate` are lower than 1.

36. Now use the results from the previous exercise and the function `which` to determine the indices of `murder_rate` associated with values lower than 1.

37. Use the results from the previous exercise to report the names of the states with murder rates lower than 1.

38. Now extend the code from the exercise to report the states in the Northeast with murder rates lower than 1. Hint: use the previously defined logical vector `low` and the logical operator `&`.

39. In a previous exercise, we computed the murder rate for each state and the average of these numbers. How many states are below the average?

40. Use the match function to identify the states with abbreviations AK, MI, and IA. Hint: start by defining an index of the entries of `murders$abb` that match the three abbreviations, then use the `[` operator to extract the states.

41. Use the `%in%` operator to create a logical vector that answers the question: which of the following are actual abbreviations: MA, ME, MI, MO, MU?

42. Extend the code you used in exercise 7 to report the one entry that is **not** an actual abbreviation. Hint: use the `!` operator, which turns `FALSE` into `TRUE` and vice versa, then `which` to obtain an index.

43. We made a plot of total murders versus population and noted a strong relationship. Not surprisingly, states with larger populations had more murders.

    ```
    population_in_millions <- murders$population/10^6
    total_gun_murders <- murders$total
    plot(population_in_millions, total_gun_murders)
    ```

    Keep in mind that many states have populations below 5 million and are bunched up. We may gain further insights from making this plot in the log scale. Transform the variables using the `log10` transformation and then plot them.

44. Create a histogram of the state populations.

45. Generate boxplots of the state populations by region.

3

Programming basics

We teach R because it greatly facilitates data analysis, the main topic of this book. By coding in R, we can efficiently perform exploratory data analysis, build data analysis pipelines, and prepare data visualization to communicate results. However, R is not just a data analysis environment but a programming language. Advanced R programmers can develop complex packages and even improve R itself, but we do not cover advanced programming in this book. Nonetheless, in this section, we introduce three key programming concepts: conditional expressions, for-loops, and functions. These are not just key building blocks for advanced programming, but are sometimes useful during data analysis. We also note that there are several functions that are widely used to program in R but that we will not cover in this book. These include `split`, `cut`, `do.call`, and `Reduce`. These are worth learning if you plan to become an expert R programmer.

3.1 Conditional expressions

Conditional expressions are one of the basic features of programming. They are used for what is called *flow control*. The most common conditional expression is the if-else statement. In R, we can actually perform quite a bit of data analysis without conditionals. However, they do come up occasionally, and you will need them once you start writing your own functions and packages.

Here is a very simple example showing the general structure of an if-else statement. The basic idea is to print the reciprocal of `a` unless `a` is 0:

```
a <- 0

if (a != 0) {
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
#> [1] "No reciprocal for 0."
```

Let's look at one more example using the US murders data frame:

```
library(dslabs)
murder_rate <- murders$total / murders$population*100000
```

DOI: 10.1201/9781003220923-3

Here is a very simple example that tells us the state with the lowest murder rate if it is lower than 0.5 per 100,000. The if-else statement protects us from the case in which no state satisfies the condition.

```
ind <- which.min(murder_rate)

if (murder_rate[ind] < 0.5) {
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
#> [1] "Vermont"
```

If we try it again with a rate of 0.25, we get a different answer:

```
if (murder_rate[ind] < 0.25) {
  print(murders$state[ind])
} else{
  print("No state has a murder rate that low.")
}
#> [1] "No state has a murder rate that low."
```

A related function that is very useful is `ifelse`. This function takes three arguments: a logical and two possible answers. If the logical is `TRUE`, the value in the second argument is returned and if `FALSE`, the value in the third argument is returned. Here is an example:

```
a <- 0
ifelse(a > 0, 1/a, NA)
#> [1] NA
```

The function is particularly useful because it works on vectors. It examines each entry of the logical vector and returns elements from the vector provided in the second argument, if the entry is `TRUE`, or elements from the vector provided in the third argument, if the entry is `FALSE`.

```
a <- c(0, 1, 2, -4, 5)
result <- ifelse(a > 0, 1/a, NA)
```

This table helps us see what happened:

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	-0.25	NA	NA
5	TRUE	0.20	NA	0.2

Here is an example of how this function can be readily used to replace all the missing values in a vector with zeros:

```
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
#> [1] 0
```

Two other useful functions are `any` and `all`. The `any` function takes a vector of logicals and returns `TRUE` if any of the entries is `TRUE`. The `all` function takes a vector of logicals and returns `TRUE` if all of the entries are `TRUE`. Here is an example:

```
z <- c(TRUE, TRUE, FALSE)
any(z)
#> [1] TRUE
all(z)
#> [1] FALSE
```

> **Note**
>
> You are ready to do exercises 1–3.

3.2 Defining functions

As you become more experienced, you will find yourself needing to perform the same operations over and over. A simple example is computing averages. We can compute the average of a vector `x` using the `sum` and `length` functions: `sum(x)/length(x)`. Because we do this repeatedly, it is much more efficient to write a function that performs this operation. This particular operation is so common that someone already wrote the `mean` function and it is included in base R. However, you will encounter situations in which the function does not already exist, so R permits you to write your own. A simple version of a function that computes the average can be defined like this:

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

Now `avg` is a function that computes the mean:

```
x <- 1:100
identical(mean(x), avg(x))
#> [1] TRUE
```

Notice that variables defined inside a function are not saved in the workspace. So while we use `s` and `n` when we call `avg`, the values are created and changed only during the call. Here is an illustrative example:

```
s <- 3
avg(1:10)
#> [1] 5.5
s
#> [1] 3
```

Note how `s` is still 3 after we call `avg`.

In general, functions are objects, so we assign them to variable names with `<-`. The function `function` tells R you are about to define a function. The general form of a function definition looks like this:

```
my_function <- function(VARIABLE_NAME){
  perform operations on VARIABLE_NAME and calculate VALUE
  VALUE
}
```

The functions you define can have multiple arguments as well as default values. For example, we can define a function that computes either the arithmetic or geometric average depending on a user-defined variable like this:

```
avg <- function(x, arithmetic = TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
```

We will learn more about how to create functions through experience as we face more complex tasks.

> **Note**
>
> You are ready to do exercises 4–7.

3.3 Namespaces

Once you start becoming more of an R expert user, you will likely need to load several add-on packages for some of your analysis. Once you start doing this, it is likely that two packages use the same name for two different functions. And often these functions do completely different things. In fact, you have already encountered this because both **dplyr** and the R-base **stats** package define a `filter` function. There are five other examples in **dplyr**. We know this because when we first load **dplyr** we see the following message:

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Now when we type `filter` it uses the **dplyr** one. But what if we want to use the **stats** version?

These functions live in different *namespaces*. R will follow a certain order when searching for a function in these *namespaces*. You can see the order by typing:

```
search()
```

The first entry in this list is the global environment which includes all the objects you define.

So what if we want to use the **stats** `filter` instead of the **dplyr** filter but **dplyr** appears first in the search list? You can force the use of a specific namespace by using double colons (`::`) like this:

```
stats::filter
```

If we want to be absolutely sure that we use the **dplyr** `filter`, we can use

```
dplyr::filter
```

Also note that if we want to use a function in a package without loading the entire package, we can use the double colon as well.

> **Tip**
>
> If you want to see all the packages that have function called, for example `filter`, you can use double questions marks: `??filter`.

For more on this more advanced topic we recommend the R packages book[1].

3.4 For-loops

The formula for the sum of the series $1 + 2 + \cdots + n$ is $S_n = n(n+1)/2$. What if we weren't sure that was the right function? How could we check? Using what we learned about functions we can create one that computes S_n:

```
compute_s_n <- function(n) {
  sum(1:n)
}
```

How can we compute S_n for various values of n, say $n = 1, \ldots, 25$? Do we write 25 lines of code calling `compute_s_n`? No, that is what for-loops are for in programming. In this case, we are performing exactly the same task over and over, and the only thing that is changing is the value of n. For-loops let us define the range that our variable takes (in our example $n = 1,\ldots, 10$), then change the value and evaluate expression as you *loop*.

[1] http://r-pkgs.had.co.nz/namespace.html

Perhaps the simplest example of a for-loop is this useless piece of code:

```
for (i in 1:5) {
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

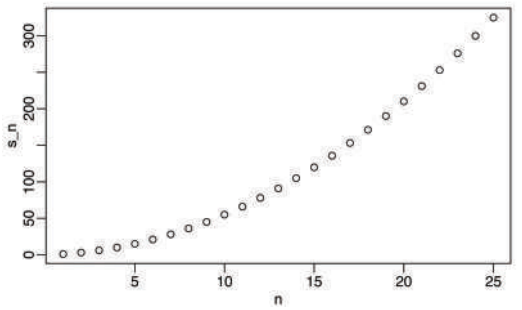
Here is the for-loop we would write for our S_n example:

```
m <- 25
s_n <- vector(length = m) # create an empty vector
for (n in 1:m) {
  s_n[n] <- compute_s_n(n)
}
```

In each iteration $n = 1$, $n = 2$, etc., we compute S_n and store it in the nth entry of `s_n`.

Now we can create a plot to search for a pattern:

```
n <- 1:m
plot(n, s_n)
```



If you noticed that it appears to be a quadratic, you are on the right track because the formula is $n(n+1)/2$.

3.5 Vectorization and functionals

Although for-loops are an important concept to understand, in R we rarely use them. As you learn more R, you will realize that *vectorization* is preferred over for-loops since it results

in shorter and clearer code. We already saw examples in the Vector Arithmetic section. A *vectorized* function is a function that will apply the same operation on each of the vectors.

```
x <- 1:10
sqrt(x)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
y <- 1:10
x*y
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

To make this calculation, there is no need for for-loops. However, not all functions work this way. For instance, the function we just wrote, `compute_s_n`, does not work element-wise since it is expecting a scalar. This piece of code does not run the function on each entry of `n`:

```
n <- 1:25
compute_s_n(n)
```

Functionals are functions that help us apply the same function to each entry in a vector, matrix, data frame, or list. Here we cover the functional that operates on numeric, logical, and character vectors: `sapply`.

The function `sapply` permits us to perform element-wise operations on any function. Here is how it works:

```
x <- 1:10
sapply(x, sqrt)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

Each element of `x` is passed on to the function `sqrt` and the result is returned. These results are concatenated. In this case, the result is a vector of the same length as the original `x`. This implies that the for-loop above can be written as follows:

```
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

Other functionals are `apply`, `lapply`, `tapply`, `mapply`, `vapply`, and `replicate`. We mostly use `sapply`, `apply`, and `replicate` in this book, but we recommend familiarizing yourselves with the others as they can be very useful.

> **Note**
>
> You are ready to do exercises 8–11.

3.6 Exercises

1. What will this conditional expression return?

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

if(all(x>0)){
  print("All Postives")
} else{
  print("Not all positives")
}
```

2. Which of the following expressions is always `FALSE` when at least one entry of a logical vector `x` is TRUE?

   (a) `all(x)`
   (b) `any(x)`
   (c) `any(!x)`
   (d) `all(!x)`

3. The function `nchar` tells you how many characters long a character vector is. Write a line of code that assigns to the object `new_names` the state abbreviation when the state name is longer than 8 characters.

4. Create a function `sum_n` that for any given value, say n, computes the sum of the integers from 1 to n (inclusive). Use the function to determine the sum of integers from 1 to 5,000.

5. Create a function `altman_plot` that takes two arguments, `x` and `y`, and plots the difference against the sum.

6. After running the code below, what is the value of `x`?

```
x <- 3
my_func <- function(y){
  x <- 5
  y+5
}
```

7. Write a function `compute_s_n` that for any given n computes the sum $S_n = 1^2 + 2^2 + 3^2 + \ldots n^2$. Report the value of the sum when $n = 10$.

8. Define an empty numerical vector `s_n` of size 25 using `s_n <- vector("numeric", 25)` and store in the results of $S_1, S_2, \ldots, S_{25}$ using a for-loop.

9. Repeat exercise 8, but this time use `sapply`.

10. Plot S_n versus n. Use points defined by $n = 1, \ldots, 25$.

11. Confirm that the formula for this sum is $S_n = n(n+1)(2n+1)/6$.

4

The tidyverse

Up to now we have been manipulating vectors by reordering and subsetting them through indexing. However, once we start more advanced analyses, the preferred unit for data storage is not the vector but the data frame. In this chapter we learn to work directly with data frames, which greatly facilitate the organization of information. We will be using data frames for the majority of this book. We will focus on a specific data format referred to as *tidy* and on specific collection of packages that are particularly helpful for working with *tidy* data referred to as the *tidyverse*.

We can load all the tidyverse packages at once by installing and loading the **tidyverse** package:

```
library(tidyverse)
```

We will learn how to implement the tidyverse approach throughout the book, but before delving into the details, in this chapter we introduce some of the most widely used tidyverse functionality, starting with the **dplyr** package for manipulating data frames and the **purrr** package for working with functions. Note that the tidyverse also includes a graphing package, **ggplot2**, which we introduce later in Chapter 8 in the Data Visualization part of the book, the **readr** package discussed in Chapter 6, and many others. In this chapter, we first introduce the concept of *tidy data* and then demonstrate how we use the tidyverse to work with data frames in this format.

4.1 Tidy data

We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations. The `murders` dataset is an example of a tidy data frame.

```
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

DOI: 10.1201/9781003220923-4

To see how the same information can be provided in different formats, consider the following example:

```
#>       country year fertility
#> 1     Germany 1960      2.41
#> 2 South Korea 1960      6.16
#> 3     Germany 1961      2.44
#> 4 South Korea 1961      5.99
#> 5     Germany 1962      2.47
#> 6 South Korea 1962      5.79
```

This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row represents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the **dslabs** package. Originally, the data was in the following format:

```
#>       country 1960 1961 1962
#> 1     Germany 2.41 2.44 2.47
#> 2 South Korea 6.16 5.99 5.79
```

The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header. For the tidyverse packages to be optimally used, data need to be reshaped into *tidy* format, which you will learn to do in the Data Wrangling part of the book. Until then, we will use example datasets that are already in tidy format.

Although not immediately obvious, as you go through the book you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

> **Note**
>
> You are ready to do exercises 1–4.

4.2 Refining data frames

The **dplyr** package from the **tidyverse** introduces functions that perform some of the most common operations when working with data frames and uses names for these functions that are relatively easy to remember. For instance, to change the data table by adding a new column, we use `mutate`. To filter the data table to a subset of rows, we use `filter`. Finally, to subset the data by selecting specific columns, we use `select`.

4.2.1 Adding columns

We want all the necessary information for our analysis to be included in the data frame. The first task is to add the murder rates to our murders data frame. The function `mutate` takes the data frame as a first argument and the name and values of the variable as a second argument using the convention `name = values`. So, to add murder rates, we use:

```
murders <- mutate(murders, rate = total/population*100000)
```

Notice that here we used `total` and `population` inside the function, which are objects that are **not** defined in our workspace. But why don't we get an error?

This is one of **dplyr**'s main features. Functions in this package, such as `mutate`, know to look for variables in the data frame provided in the first argument. In the call to mutate above, `total` will have the values in `murders$total`. This approach makes the code much more readable.

We can see that the new column is added:

```
head(murders)
#>        state abb region population total rate
#> 1    Alabama  AL  South    4779736   135 2.82
#> 2     Alaska  AK   West     710231    19 2.68
#> 3    Arizona  AZ   West    6392017   232 3.63
#> 4   Arkansas  AR  South    2915918    93 3.19
#> 5 California  CA   West   37253956  1257 3.37
#> 6   Colorado  CO   West    5029196    65 1.29
```

4.2.2 Row-wise subsetting

Now suppose that we want to filter the data frame to only show the entries for which the murder rate is lower than 0.71. To do this we use the `filter` function, which takes the data frame as the first argument and then a conditional statement as the second. Like `mutate`, we can use the unquoted variable names from `murders` inside the function and it will know we mean the columns and not objects in the workspace.

```
filter(murders, rate <= 0.71)
#>           state abb        region population total  rate
#> 1        Hawaii  HI          West    1360301     7 0.515
#> 2          Iowa  IA North Central    3046355    21 0.689
#> 3 New Hampshire  NH     Northeast    1316470     5 0.380
#> 4  North Dakota  ND North Central     672591     4 0.595
#> 5       Vermont  VT     Northeast     625741     2 0.320
```

4.2.3 Column-wise subsetting

Although our data frame only has six columns, some data frames include hundreds. If we want to view just a few, we can use the **dplyr** `select` function. In the code below we select three columns, assign this to a new object and then filter the new object:

```
new_dataframe <- select(murders, state, region, rate)
filter(new_dataframe, rate <= 0.71)
#>           state        region  rate
#> 1        Hawaii          West 0.515
#> 2          Iowa North Central 0.689
#> 3 New Hampshire     Northeast 0.380
```

```
#> 4  North Dakota North Central 0.595
#> 5       Vermont     Northeast 0.320
```

In the call to `select`, the first argument `murders` is an object, but `state`, `region`, and `rate` are variable names.

dplyr offers a series of helper functions to select columns based on their content. For example, the following code uses the function `where` to keep only the numeric columns:

```
new_dataframe <- select(murders, where(is.numeric))
names(new_dataframe)
#> [1] "population" "total"      "rate"
```

The helper functions `starts_with`, `ends_with`, `contains`, `matches`, and `num_range` can be used to select columns based on their names. Here is an example showing all the rows that start with `r`:

```
new_dataframe <- select(murders, starts_with("r"))
names(new_dataframe)
#> [1] "region" "rate"
```

The helper function `everything` selects all columns.

4.2.4 Transforming variables

The function `mutate` can also be used to transform variables. For example, the following code takes the log transformation of the population variable:

```
mutate(murders, population = log10(population))
```

Often, we need to apply the same transformation to several variables. The function `across` facilitates the operation. For example if want to log transform both population and total murders we can use:

```
mutate(murders, across(c(population, total), log10))
```

The helper functions come in handy when using across. An example is if we want to apply the same transformation to all numeric variables:

```
mutate(murders, across(where(is.numeric), log10))
```

or all character variables:

```
mutate(murders, across(where(is.character), tolower))
```

> **Note**
>
> You are ready to do exercises 5–11.

4.3 The pipe

In R we can perform a series of operations, for example `select` and then `filter`, by sending the results of one function to another using what is called the *pipe operator*: `%>%`. Since R version 4.1.0, you can also use `|>`. Some details are included below.

We wrote code in Section 4.2.3 to show three variables (state, region, rate) for states that have murder rates below 0.71. To do this, we defined the intermediate object `new_dataframe`. In **dplyr** we can write code that looks more like a description of what we want to do without intermediate objects:

original data $\rightarrow$ select $\rightarrow$ filter

For such an operation, we can use the pipe `|>`. The code looks like this:

```
murders |> select(state, region, rate) |> filter(rate <= 0.71)
#>           state        region  rate
#> 1        Hawaii          West 0.515
#> 2          Iowa North Central 0.689
#> 3 New Hampshire     Northeast 0.380
#> 4  North Dakota North Central 0.595
#> 5       Vermont     Northeast 0.320
```

This line of code is equivalent to the two lines in Section 4.2.3. What is going on here?

In general, the pipe *sends* the result of the left side of the pipe to be the first argument of the function on the right side of the pipe. Here is a very simple example:

```
16 |> sqrt()
#> [1] 4
```

We can continue to pipe values along:

```
16 |> sqrt() |> log2()
#> [1] 2
```

The above statement is equivalent to `log2(sqrt(16))`.

Remember that the pipe sends values to the first argument, so we can define other arguments as if the first argument is already defined:

```
16 |> sqrt() |> log(base = 2)
#> [1] 2
```

Therefore, when using the pipe with data frames and **dplyr**, we no longer need to specify the required first argument since the **dplyr** functions we have described all take the data as the first argument. In the code we wrote:

```
murders |> select(state, region, rate) |> filter(rate <= 0.71)edsfsdf
```

`murders` is the first argument of the `select` function, and the new data frame (formerly `new_dataframe`) is the first argument of the `filter` function.

Note that the pipe works well with functions where the first argument is the input data. Functions in **tidyverse** packages like **dplyr** have this format and can be used easily with the pipe.

4.4 Summarizing data

An important part of exploratory data analysis is summarizing data. The average and standard deviation are two examples of widely used summary statistics. More informative summaries can often be achieved by first splitting data into groups. In this section, we cover two new **dplyr** verbs that make these computations easier: `summarize` and `group_by`. We learn to access resulting values using the `pull` function.

4.4.1 The `summarize` function

The `summarize` function in **dplyr** provides a way to compute summary statistics with intuitive and readable code. We start with a simple example based on heights. The `heights` dataset includes heights and sex reported by students in an in-class survey.

```
library(dplyr)
library(dslabs)
```

The following code computes the average and standard deviation for females:

```
s <- heights |>
  filter(sex == "Female") |>
  summarize(average = mean(height), standard_deviation = sd(height))
s
#>   average standard_deviation
#> 1    64.9               3.76
```

This takes our original data frame as input, filters it to keep only females, and then produces a new summarized table with just the average and the standard deviation of heights. We get to choose the names of the columns of the resulting table. For example, above we decided to use `average` and `standard_deviation`, but we could have used other names just the same.

Because the resulting table stored in `s` is a data frame, we can access the components with the accessor `$`:

```
s$average
#> [1] 64.9
s$standard_deviation
#> [1] 3.76
```

As with most other **dplyr** functions, `summarize` is aware of the variable names and we can use them directly. So when inside the call to the `summarize` function we write `mean(height)`,

the function is accessing the column with the name "height" and then computing the average of the resulting numeric vector. We can compute any other summary that operates on vectors and returns a single value.

For another example of how we can use the `summarize` function, let's compute the average murder rate for the United States. Remember our data table includes total murders and population size for each state and we have already used **dplyr** to add a murder rate column:

```
murders <- murders |> mutate(rate = total/population*100000)
```

Remember that the US murder rate is **not** the average of the state murder rates:

```
murders |>
  summarize(rate = mean(rate))
#>   rate
#> 1 2.78
```

This is because in the computation above, the small states are given the same weight as the large ones. The US murder rate is the total number of murders in the US divided by the total US population. So the correct computation is:

```
us_murder_rate <- murders |>
  summarize(rate = sum(total)/sum(population)*100000)
us_murder_rate
#>   rate
#> 1 3.03
```

This computation counts larger states proportionally to their size which results in a larger value.

4.4.2 Multiple summaries

Suppose we want three summaries from the same variable such as the median, minimum, and maximum heights. We can use `summarize` like this:

```
heights |> summarize(median = median(height), min = min(height),
        max = max(height))
#>   median min  max
#> 1   68.5  50 82.7
```

But we can obtain these three values with just one line using the `quantile` function: `quantile(x, c(0.5, 0, 1))` returns the median (50th percentile), the min (0th percentile), and max (100th percentile) of the vector `x`. Here we can't use `summarize` because it expects one value per row. Instead we have to use the `reframe` function:

```
heights |> reframe(quantiles = quantile(height, c(0.5, 0, 1)))
#>   quantiles
#> 1      68.5
#> 2      50.0
#> 3      82.7
```

However, if we want a column per summary, as the `summarize` call above, we have to define a function that returns a data frame like this:

```
median_min_max <- function(x){
  qs <- quantile(x, c(0.5, 0, 1))
  data.frame(median = qs[1], min = qs[2], max = qs[3])
}
```

Then we can call `summarize` as above:

```
heights |> summarize(median_min_max(height))
#>   median min  max
#> 1   68.5  50 82.7
```

4.4.3 Group then summarize with `group_by`

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. The `group_by` function helps us do this.

If we type this:

```
heights |> group_by(sex)
#> # A tibble: 1,050 x 2
#> # Groups:   sex [2]
#>   sex    height
#>   <fct>  <dbl>
#> 1 Male      75
#> 2 Male      70
#> 3 Male      68
#> 4 Male      74
#> 5 Male      61
#> # i 1,045 more rows
```

The result does not look very different from `heights`, except we see `Groups: sex [2]` when we print the object. Although not immediately obvious from its appearance, this is now a special data frame called a *grouped data frame*, and **dplyr** functions, in particular `summarize`, will behave differently when acting on this object. Conceptually, you can think of this table as many tables, with the same columns but not necessarily the same number of rows, stacked together in one object. When we summarize the data after grouping, this is what happens:

```
heights |>
  group_by(sex) |>
  summarize(average = mean(height), standard_deviation = sd(height))
#> # A tibble: 2 x 3
#>   sex    average standard_deviation
#>   <fct>    <dbl>              <dbl>
#> 1 Female    64.9               3.76
#> 2 Male      69.3               3.61
```

The `summarize` function applies the summarization to each group separately.

For another example, let's compute the median, minimum, and maximum murder rate in the four regions of the country using the `median_min_max` defined above:

```
murders |>
  group_by(region) |>
  summarize(median_min_max(rate))
#> # A tibble: 4 x 4
#>   region        median   min   max
#>   <fct>          <dbl> <dbl> <dbl>
#> 1 Northeast       1.80 0.320  3.60
#> 2 South           3.40 1.46  16.5
#> 3 North Central   1.97 0.595  5.36
#> 4 West            1.29 0.515  3.63
```

4.4.4 Extracting variables with `pull`

The `us_murder_rate` object defined in Section 4.4.1 represents just one number. Yet we are storing it in a data frame:

```
class(us_murder_rate)
#> [1] "data.frame"
```

since, as most **dplyr** functions, `summarize` always returns a data frame.

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the `pull` function. To get a number from the original data table with one additional line of code we can type:

```
us_murder_rate <- murders |>
  summarize(rate = sum(total)/sum(population)*100000) |>
  pull(rate)

us_murder_rate
#> [1] 3.03
```

which is now a numeric:

```
class(us_murder_rate)
#> [1] "numeric"
```

4.5 Sorting

When examining a dataset, it is often convenient to sort the data frame by the different columns. We know about the `order` and `sort` function, but for ordering entire data frames,

the **dplyr** function `arrange` is useful. For example, here we order the states by population size:

```
murders |> arrange(population) |> head()
#>                  state abb        region population total   rate
#> 1              Wyoming  WY          West     563626     5  0.887
#> 2 District of Columbia  DC         South     601723    99 16.453
#> 3              Vermont  VT     Northeast     625741     2  0.320
#> 4         North Dakota  ND North Central     672591     4  0.595
#> 5               Alaska  AK          West     710231    19  2.675
#> 6         South Dakota  SD North Central     814180     8  0.983
```

With `arrange` we get to decide which column to sort by. To see the states sorted by murder rates, for example, we would use `arrange(rate)` instead.

Note that the default behavior is to order in ascending order. In **dplyr**, the function `desc` transforms a vector so that it is in descending order. To sort the data frame by murder rates in descending order, we can type:

```
murders |> arrange(desc(rate))
```

4.5.1 Nested sorting

If we are ordering by a column with ties, we can use a second column to break the tie. Similarly, a third column can be used to break ties between first and second and so on. Here we order by `region`, then within region we order by murder rate:

```
murders |>
  arrange(region, rate) |>
  head()
#>           state abb    region population total  rate
#> 1       Vermont  VT Northeast     625741     2 0.320
#> 2 New Hampshire  NH Northeast    1316470     5 0.380
#> 3         Maine  ME Northeast    1328361    11 0.828
#> 4  Rhode Island  RI Northeast    1052567    16 1.520
#> 5 Massachusetts  MA Northeast    6547629   118 1.802
#> 6      New York  NY Northeast   19378102   517 2.668
```

4.5.2 The top n

In the code above, we have used the function `head` to avoid having the page fill up with the entire dataset. For instance, using `arrange(desc(rate))` followed by `head` would show the 6 states with the largest murder rates, in order. Instead, to view a specific number of observations with the highest murder rates, we can use the `top_n` function. This function takes a data frame as it's first argument, the number of rows to show in the second, and the variable to filter by in the third. Here is an example of how to see the top 5 rows:

```
murders |> top_n(5, rate)
#>                  state abb        region population total  rate
#> 1 District of Columbia  DC         South     601723    99 16.45
#> 2            Louisiana  LA         South    4533372   351  7.74
#> 3             Maryland  MD         South    5773552   293  5.07
#> 4             Missouri  MO North Central    5988927   321  5.36
#> 5       South Carolina  SC         South    4625364   207  4.48
```

Note that rows are not sorted by `rate`, only filtered. If we want to sort, we still need to use `arrange`. Note that if the third argument is left blank, `top_n` filters by the last column.

> **Note**
>
> You are ready to do exercises 12–19.

4.6 Tibbles

To work with the tidyverse, data must be stored in data frames. We introduced the data frame in Section 2.3.1 and have been using the `murders` data frame throughout the book. In Section 4.4.3 we introduced the `group_by` function, which permits stratifying data before computing summary statistics. But where is the group information stored in the data frame?

```
murders |> group_by(region)
#> # A tibble: 51 x 6
#> # Groups:   region [4]
#>   state      abb   region population total  rate
#>   <chr>      <chr> <fct>       <dbl> <dbl> <dbl>
#> 1 Alabama    AL    South     4779736   135  2.82
#> 2 Alaska     AK    West       710231    19  2.68
#> 3 Arizona    AZ    West      6392017   232  3.63
#> 4 Arkansas   AR    South     2915918    93  3.19
#> 5 California CA    West     37253956  1257  3.37
#> # i 46 more rows
```

Notice that there are no columns with this information. But, if you look closely at the output above, you see the line `A tibble` followed by dimensions. We can learn the class of the returned object using:

```
murders |> group_by(region) |> class()
#> [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

The `tbl`, pronounced "tibble", is a special kind of data frame. The functions `group_by` and `summarize` always return this type of data frame. The `group_by` function returns a special kind of `tbl`, the `grouped_df`. We will say more about these later. For consistency, the **dplyr** manipulation verbs (`select`, `filter`, `mutate`, and `arrange`) preserve the class of the input: if they receive a regular data frame they return a regular data frame, while if they receive a tibble they return a tibble. But tibbles are the preferred format in the tidyverse and as

a result tidyverse functions that produce a data frame from scratch return a tibble. For example, in Chapter 6 we will see that tidyverse functions used to import data create tibbles.

4.7 Tibbles versus data frames

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames. Nonetheless there are some important differences which we describe next.

(1) Tibbles display better

The print method for tibbles is more readable than that of a data frame. To see this, compare the outputs of typing `murders` and the output of murders if we convert it to a tibble. We can do this using `as_tibble(murders)`. If using RStudio, output for a tibble adjusts to your window size. To see this, change the width of your R console and notice how more/less columns are shown.

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

```
class(murders[,4])
#> [1] "numeric"
```

is not a data frame. With tibbles this does not happen:

```
class(as_tibble(murders)[,4])
#> [1] "tbl_df"     "tbl"        "data.frame"
```

This is useful in the tidyverse since functions require data frames as input.

With tibbles, if you want to access the vector that defines a column, and not get back a data frame, you need to use the accessor `$`:

```
class(as_tibble(murders)$population)
#> [1] "numeric"
```

A related feature is that tibbles will give you a warning if you try to access a column that does not exist. If we accidentally write `Population` instead of `population` this:

```
murders$Population
#> NULL
```

returns a `NULL` with no warning, which can make it harder to debug. In contrast, if we try this with a tibble we get an informative warning:

```
as_tibble(murders)$Population
#> Warning: Unknown or uninitialised column: `Population`.
#> NULL
```

(2) Tibbles can have complex entries

While data frame columns need to be vectors of numbers, strings, or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

```
tibble(id = c(1, 2, 3), func = c(mean, median, sd))
#> # A tibble: 3 x 2
#>      id func
#>   <dbl> <list>
#> 1     1 <fn>
#> 2     2 <fn>
#> 3     3 <fn>
```

(3) Tibbles can be grouped

The function `group_by` returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the `summarize` function, are aware of the group information.

4.7.1 Creating tibbles

It is sometimes useful for us to create our own data frames. To create a data frame in the tibble format, you can do this by using the `tibble` function.

```
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
                 exam_1 = c(95, 80, 90, 85),
                 exam_2 = c(90, 85, 85, 90))
```

Note that base R (without packages loaded) has the `data.frame` function that can be used to create a regular data frame rather than a tibble.

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
```

To convert a regular data frame to a tibble, you can use the `as_tibble` function.

```
as_tibble(grades) |> class()
#> [1] "tbl_df"     "tbl"        "data.frame"
```

4.8 The placeholder

One of the advantages of using the pipe `|>` is that we do not have to keep naming new objects as we manipulate the data frame. The object on the left-hand side of the pipe is used as the first argument of the function on the right-hand side of the pipe. But what if we want to pass it as argument to the right-hand side function that is not the first? The answer is

the placeholder operator `_` (for the `%>%` pipe the placeholder is `.`). Below is a simple example that passes the `base` argument to the `log` function. The following three are equivalent:

```
log(8, base = 2)
2 |> log(8, base = _)
2 %>% log(8, base = .)
```

4.9 The purrr package

In Section 3.5 we learned about the `sapply` function, which permitted us to apply the same function to each element of a vector. We constructed a function and used `sapply` to compute the sum of the first `n` integers for several values of `n` like this:

```
compute_s_n <- function(n) {
  sum(1:n)
}
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

This type of operation, applying the same function or procedure to elements of an object, is quite common in data analysis. The **purrr** package includes functions similar to `sapply` but that better interact with other tidyverse functions. The main advantage is that we can better control the output type of functions. In contrast, `sapply` can return several different object types; for example, we might expect a numeric result from a line of code, but `sapply` might convert our result to character under some circumstances. **purrr** functions will never do this: they will return objects of a specified type or return an error if this is not possible.

The first **purrr** function we will learn is `map`, which works very similar to `sapply` but always, without exception, returns a list:

```
library(purrr)
s_n <- map(n, compute_s_n)
class(s_n)
#> [1] "list"
```

If we want a numeric vector, we can instead use `map_dbl` which always returns a vector of numeric values.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
#> [1] "numeric"
```

This produces the same results as the `sapply` call shown above.

A particularly useful **purrr** function for interacting with the rest of the tidyverse is `map_df`, which always returns a tibble data frame. However, the function being called needs to return a vector or a list with names. For this reason, the following code would result in a `Argument 1 must have names` error:

```
s_n <- map_df(n, compute_s_n)
```

The function needs to return a data frame to make this work:

```
compute_s_n <- function(n) {
  tibble(sum = sum(1:n))
}
s_n <- map_df(n, compute_s_n)
```

The **purrr** package provides much more functionality not covered here. For more details you can consult this online resource[1].

4.10 Tidyverse conditionals

A typical data analysis will often involve one or more conditional operations. In Section 3.1 we described the `ifelse` function, which we will use extensively in this book. In this section we present two **dplyr** functions that provide further functionality for performing conditional operations.

4.10.1 `case_when`

The `case_when` function is useful for vectorizing conditional statements. It is similar to `ifelse` but can output any number of values, as opposed to just `TRUE` or `FALSE`. Here is an example splitting numbers into negative, positive, and 0:

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative",
          x > 0 ~ "Positive",
          TRUE  ~ "Zero")
#> [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

A common use for this function is to define categorical variables based on existing variables. For example, suppose we want to compare the murder rates in four groups of states: *New England*, *West Coast*, *South*, and *other*. For each state, we need to ask if it is in New England, if it is not we ask if it is in the West Coast, if not we ask if it is in the South, and if not we assign *other*. Here is how we use `case_when` to do this:

```
murders |>
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) |>
  group_by(group) |>
```

[1] https://jennybc.github.io/purrr-tutorial/

```
  summarize(rate = sum(total)/sum(population)*10^5)
#> # A tibble: 4 x 2
#>   group        rate
#>   <chr>       <dbl>
#> 1 New England  1.72
#> 2 Other        2.71
#> 3 South        3.63
#> 4 West Coast   2.90
```

4.10.2 `between`

A common operation in data analysis is to determine if a value falls inside an interval. We can check this using conditionals. For example, to check if the elements of a vector `x` are between `a` and `b` we can type

```
x >= a & x <= b
```

However, this can become cumbersome, especially within the tidyverse approach. The `between` function performs the same operation.

```
between(x, a, b)
```

4.11 Exercises

1. Examine the built-in dataset `co2`. Which of the following is true:

   (a) `co2` is tidy data: it has one year for each row.
   (b) `co2` is not tidy: we need at least one column with a character vector.
   (c) `co2` is not tidy: it is a matrix instead of a data frame.
   (d) `co2` is not tidy: to be tidy we would have to wrangle it to have three columns (year, month and value), then each co2 observation would have a row.

2. Examine the built-in dataset `ChickWeight`. Which of the following is true:

   (a) `ChickWeight` is not tidy: each chick has more than one row.
   (b) `ChickWeight` is tidy: each observation (a weight) is represented by one row. The chick from which this measurement came is one of the variables.
   (c) `ChickWeight` is not tidy: we are missing the year column.
   (d) `ChickWeight` is tidy: it is stored in a data frame.

3. Examine the built-in dataset `BOD`. Which of the following is true:

   (a) `BOD` is not tidy: it only has six rows.
   (b) `BOD` is not tidy: the first column is just an index.
   (c) `BOD` is tidy: each row is an observation with two values (time and demand)
   (d) `BOD` is tidy: all small datasets are tidy by definition.

4. Which of the following built-in datasets is tidy (you can pick more than one):

   (a) `BJsales`
   (b) `EuStockMarkets`
   (c) `DNase`
   (d) `Formaldehyde`
   (e) `Orange`
   (f) `UCBAdmissions`

5. Load the **dplyr** package and the murders dataset.

   ```
   library(dplyr)
   library(dslabs)
   ```

   You can add columns using the **dplyr** function `mutate`. This function is aware of the column names and inside the function you can call them unquoted:

   ```
   murders <- mutate(murders, population_in_millions = population/10^6)
   ```

   We can write `population` rather than `murders$population`. The function `mutate` knows we are grabbing columns from `murders`.

   Use the function `mutate` to add a murders column named `rate` with the per 100,000 murder rate as in the example code above. Make sure you redefine `murders` as done in the example code above (`murders <- [your code]`) so we can keep using this variable.

6. If `rank(x)` gives you the ranks of `x` from lowest to highest, `rank(-x)` gives you the ranks from highest to lowest. Use the function `mutate` to add a column `rank` containing the rank of murder rate from highest to lowest. Make sure you redefine `murders` so we can keep using this variable.

7. With **dplyr**, we can use `select` to show only certain columns. For example, with this code we would only show the states and population sizes:

   ```
   select(murders, state, population)
   ```

   Use `select` to show the state names and abbreviations in `murders`. Do not redefine `murders`, just show the results.

8. The **dplyr** function `filter` is used to choose specific rows of the data frame to keep. Unlike `select` which is for columns, `filter` is for rows. For example, you can show just the New York row like this:

   ```
   filter(murders, state == "New York")
   ```

   You can use other logical vectors to filter rows.

   Use `filter` to show the top 5 states with the highest murder rates. From here on, do not change the murders dataset, just show the result. Remember that you can filter based on the `rank` column.

9. We can remove rows using the `!=` operator. For example, to remove Florida, we would do this:

```
no_florida <- filter(murders, state != "Florida")
```

Create a new data frame called `no_south` that removes states from the South region. How many states are in this category? You can use the function `nrow` for this.

10. We can also use `%in%` to filter with **dplyr**. You can therefore see the data from New York and Texas like this:

```
filter(murders, state %in% c("New York", "Texas"))
```

Create a new data frame called `murders_nw` with only the states from the Northeast and the West. How many states are in this category?

11. Suppose you want to live in the Northeast or West **and** want the murder rate to be less than 1. We want to see the data for the states satisfying these options. Note that you can use logical operators like `&` with `filter`. Here is an example in which we filter to keep only small states in the Northeast region.

```
filter(murders, population < 5000000 & region == "Northeast")
```

Make sure `murders` has been defined with `rate` and `rank` and still has all states. Create a table called `my_states` that contains rows for states satisfying both the conditions: it is in the Northeast or West and the murder rate is less than 1. Use `select` to show only the state name, the rate, and the rank.

12. The pipe `|>` can be used to perform operations sequentially without having to define intermediate objects. Start by redefining `murders` to include rate and rank.

```
murders <- mutate(murders, rate =  total/population*100000,
                  rank = rank(-rate))
```

In the solution to the previous exercise, we did the following:

```
my_states <- filter(murders, region %in% c("Northeast", "West") &
                      rate < 1)

select(my_states, state, rate, rank)
```

The pipe `|>` permits us to perform both operations sequentially without having to define an intermediate variable `my_states`. We therefore could have mutated and selected in the same line like this:

```
mutate(murders, rate =  total/population*100000,
       rank = rank(-rate)) |>
  select(state, rate, rank)
```

Notice that `select` no longer has a data frame as the first argument. The first argument is assumed to be the result of the operation conducted right before the `|>`.

Repeat the previous exercise, but now instead of creating a new object, show the result and only include the state, rate, and rank columns. Use a pipe `|>` to do this in just one line.

```
my_states <- murders |>
  mutate SOMETHING |>
  filter SOMETHING |>
  select SOMETHING
```

13. For exercises 13–19, we will be using the data from the survey collected by the United States National Center for Health Statistics (NCHS). This center has conducted a series of health and nutrition surveys since the 1960s. Starting in 1999, about 5,000 individuals of all ages have been interviewed every year and they complete the health examination component of the survey. Part of the data is made available via the **NHANES** package. Once you install the **NHANES** package, you can load the data like this:

```
library(NHANES)
```

The **NHANES** data has many missing values. The `mean` and `sd` functions in R will return `NA` if any of the entries of the input vector is an `NA`. Here is an example:

```
library(dslabs)
mean(na_example)
#> [1] NA
sd(na_example)
#> [1] NA
```

To ignore the `NA`s we can use the `na.rm` argument:

```
mean(na_example, na.rm = TRUE)
#> [1] 2.3
sd(na_example, na.rm = TRUE)
#> [1] 1.22
```

Let's now explore the NHANES data.

We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females. `AgeDecade` is a categorical variable with these ages. Note that the category is coded like `" 20-29"`, with a space in front! What is the average and standard deviation of systolic blood pressure as saved in the `BPSysAve` variable? Save it to a variable called `ref`.

Hint: Use `filter` and `summarize` and use the `na.rm = TRUE` argument when computing the average and standard deviation. You can also filter the NA values using `filter`.

14. Using a pipe, assign the average to a numeric variable `ref_avg`. Hint: Use the code from the previous exercise and then `pull`.

15. Now report the min and max values for the same group.

16. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in exercise 13. Note that the age groups are defined by `AgeDecade`. Hint: rather than filtering by age and gender, filter by `Gender` and then use `group_by`.

17. Repeat exercise 16 for males.

18. For males between the ages of 40–49, compare systolic blood pressure across race as reported in the `Race1` variable. Order the resulting table from lowest to highest average systolic blood pressure.

19. Load the `murders` dataset. Which of the following is true?

    (a) `murders` is in tidy format and is stored in a tibble.
    (b) `murders` is in tidy format and is stored in a data frame.
    (c) `murders` is not in tidy format and is stored in a tibble.
    (d) `murders` is not in tidy format and is stored in a data frame.

20. Use `as_tibble` to convert the `murders` data table into a tibble and save it in an object called `murders_tibble`.

21. Use the `group_by` function to convert `murders` into a tibble that is grouped by region.

22. Write tidyverse code that is equivalent to this code:

    ```
    exp(mean(log(murders$population)))
    ```

    Write it using the pipe so that each function is called without arguments. Use the dot operator to access the population. Hint: The code should start with `murders |>`.

23. Use the `map_df` to create a data frame with three columns named `n`, `s_n`, and `s_n_2`. The first column should contain the numbers 1 through 100. The second and third columns should each contain the sum of 1 through n with n the row number.

5

data.table

In this book, we use tidyverse packages, primarily because they offer readability that is beneficial for beginners. This readability allows us to emphasize data analysis and statistical concepts. However, while tidyverse is beginner-friendly, there are other methods in R that are more efficient and can handle larger datasets more effectively. One such package is **data.table**, which is widely used in the R community. We'll briefly introduce **data.table** in this chapter. For those interested in diving deeper, there are numerous online resources, including the mentioned introduction[1].

5.1 Refining data tables

`data.table` is a separate package that needs to be installed. Once installed, we then need to load it along with the other packages we will use:

```
library(dplyr)
library(dslabs)
library(data.table)
```

We will provide example code showing the **data.table** approaches to **dplyr**'s `mutate`, `filter`, `select`, `group_by`, and `summarize` shown in Chapter 4. As in that chapter, we will use the `murders` dataset:

The first step when using **data.table** is to convert the data frame into a `data.table` object using the `as.data.table` function:

```
murders_dt <- as.data.table(murders)
```

Without this initial step, most of the approaches shown below will not work.

5.1.1 Column-wise subsetting

Selecting with **data.table** is done in a similar way to subsetting matrices. While with **dplyr** we write

```
select(murders, state, region)
```

[1] https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

DOI: 10.1201/9781003220923-5

in **data.table** we use

```
murders_dt[, c("state", "region")]
```

We can also use the `.()` **data.table** notation to alert R that variables inside the parenthesis are column names, not objects in the R environment. So the above can also be written like this:

```
murders_dt[, .(state, region)]
```

5.1.2 Adding or transformin variables

We learned to use the **dplyr** `mutate` function with this example:

```
murders <- mutate(murders, rate = total / population * 100000)
```

data.table uses an approach that avoids a new assignment (update by reference). This can help with large datasets that take up most of your computer's memory. The **data.table** `:=` function permits us to do this:

```
murders_dt[, rate := total / population * 100000]
```

This adds a new column, `rate`, to the table. Notice that, as in **dplyr**, we used `total` and `population` without quotes.

To define new multiple columns, we can use the `:=` function with multiple arguments:

```
murders_dt[, ":="(rate = total / population * 100000,
        rank = rank(population))]
```

5.1.3 Reference versus copy

The **data.table** package is designed to avoid wasting memory. So if you make a copy of a table, like this:

```
x <- data.table(a = 1)
y <- x
```

`y` is actually referencing `x`, it is not an new object: `y` just another name for `x`. Until you change `y`, a new object will not be made. However, the `:=` function changes *by reference* so if you change `x`, a new object is not made and `y` continues to be just another name for `x`:

```
x[,a := 2]
y
#>    a
#> 1: 2
```

You can also change `x` like this:

```
y[,a := 1]
x
#>    a
#> 1: 1
```

To avoid this, you can use the `copy` function which forces the creation of an actual copy:

```
x <- data.table(a = 1)
y <- copy(x)
x[,a := 2]
y
#>    a
#> 1: 1
```

Note that the function `as.data.table` creates a copy of the data frame being converted. However, if working with a large data frames it is helpful to avoid this by using `setDT`:

```
x <- data.frame(a = 1)
setDT(x)
```

Note that because no copy is being made the following code does not create a new object:

```
x <- data.frame(a = 1)
y <- setDT(x)
```

The objects `x` and `y` are referencing the same data table:

```
x[,a := 2]
y
#>    a
#> 1: 2
```

5.1.4 Row-wise subsetting

With **dplyr**, we filtered like this:

```
filter(murders, rate <= 0.7)
```

With **data.table**, we again use an approach similar to subsetting matrices, except like **dplyr**, **data.table** knows that `rate` refers to a column name and not an object in the R environment:

```
murders_dt[rate <= 0.7]
```

Notice that we can combine the filter and select into one succinct command. Here are the state names and rates for those with rates below 0.7.

```
murders_dt[rate <= 0.7, .(state, rate)]
#>            state  rate
#> 1:        Hawaii 0.515
#> 2:          Iowa 0.689
#> 3: New Hampshire 0.380
#> 4:  North Dakota 0.595
#> 5:       Vermont 0.320
```

which is more compact than the **dplyr** approach:

```
murders |> filter(rate <= 0.7) |> select(state, rate)
```

> **Note**
>
> You are ready to do exercises 1–7.

5.2 Summarizing data

As an example, we will use the `heights` dataset:

```
heights_dt <- as.data.table(heights)
```

In **data.table**, we can call functions inside `.()` and they will be applied to columns So the equivalent of:

```
s <- heights |> summarize(avg = mean(height), sd = sd(height))
```

in **dplyr** is the following in **data.table**:

```
s <- heights_dt[, .(avg = mean(height), sd = sd(height))]
```

Note that this permits a compact way of subsetting and then summarizing. Instead of:

```
s <- heights |>
  filter(sex == "Female") |>
  summarize(avg = mean(height), sd = sd(height))
```

we can write:

```
s <- heights_dt[sex == "Female", .(avg = mean(height), sd = sd(height))]
```

5.2.1 Multiple summaries

In Chapter 4, we defined the following function to permit multiple column summaries in **dplyr**:

```
median_min_max <- function(x){
  qs <- quantile(x, c(0.5, 0, 1))
  data.frame(median = qs[1], minimum = qs[2], maximum = qs[3])
}
```

In **data.table** we place a function call within `.()` to obtain the three number summary:

```
heights_dt[, .(median_min_max(height))]
```

5.2.2 Group then summarize

The `group_by` followed by `summarize` in **dplyr** is performed in one line in **data.table**. We simply add the `by` argument to split the data into groups based on the values in categorical variable:

```
heights_dt[, .(avg = mean(height), sd = sd(height)), by = sex]
#>       sex  avg   sd
#> 1:   Male 69.3 3.61
#> 2: Female 64.9 3.76
```

5.3 Sorting

We can order rows using the same approach we use for filter. Here are the states ordered by murder rate:

```
murders_dt[order(population)]
```

To sort the table in descending order, we can order by the negative of `population` or use the `decreasing` argument:

```
murders_dt[order(population, decreasing = TRUE)]
```

Similarly, we can perform nested ordering by including more than one variable in order:

```
murders_dt[order(region, rate)]
```

> **Note**
>
> You are ready to do exercises 8–12.

5.4 Exercises

1. Load the **data.table** package and the `murders` dataset and convert it to `data.table` object:

   ```
   library(data.table)
   library(dslabs)
   murders_dt <- as.data.table(murders)
   ```

   Remember you can add columns like this:

   ```
   murders_dt[, population_in_millions := population / 10^6]
   ```

   Add a `murders` column named `rate` with the per 100,000 murder rate as in the example code above.

2. Add a column `rank` containing the rank, from highest to lowest murder rate.

3. If we want to only show the states and population sizes, we can use:

   ```
   murders_dt[, .(state, population)]
   ```

   Show the state names and abbreviations in `murders`.

4. You can show just the New York row like this:

   ```
   murders_dt[state == "New York"]
   ```

   You can use other logical vectors to filter rows.

   Show the top 5 states with the highest murder rates. From here on, do not change the `murders` dataset, just show the result. Remember that you can filter based on the `rank` column.

5. We can remove rows using the `!=` operator. For example, to remove Florida, we would do this:

   ```
   no_florida <- murders_dt[state != "Florida"]
   ```

   Create a new data frame called `no_south` that removes states from the South region. How many states are in this category? You can use the function `nrow` for this.

6. We can also use `%in%` to filter. You can therefore see the data from New York and Texas as follows:

   ```
   murders_dt[state %in% c("New York", "Texas")]
   ```

   Create a new data frame called `murders_nw` with only the states from the Northeast and the West. How many states are in this category?

7. Suppose you want to live in the Northeast or West **and** want the murder rate to be less than 1. We want to see the data for the states satisfying these options. Note that you can use logical operators with `filter`. Here is an example in which we filter to keep only small states in the Northeast region.

```
murders_dt[population < 5000000 & region == "Northeast"]
```

   Make sure `murders` has been defined with `rate` and `rank` and still has all states. Create a table called `my_states` that contains rows for states satisfying both the conditions: they are in the Northeast or West and the murder rate is less than 1. Show only the state name, the rate, and the rank.

   For exercises 8–12, we will be using the **NHANES** data.

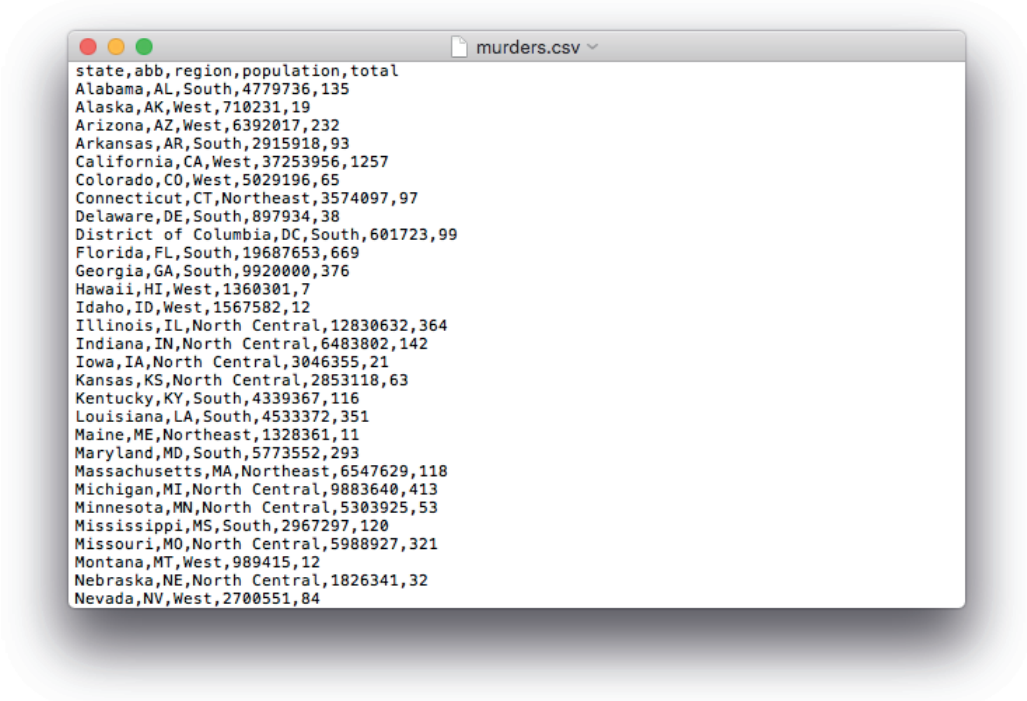
```
library(NHANES)
```

8. We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females. `AgeDecade` is a categorical variable with these ages. Note that the category is coded like `" 20-29"`, with a space in front! Use the **data.table** package to compute the average and standard deviation of systolic blood pressure as saved in the `BPSysAve` variable. Save it to a variable called `ref`.
9. Report the min and max values for the same group.
10. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in exercise 8. Note that the age groups are defined by `AgeDecade`.
11. Repeat exercise 10 for males.
12. For males between the ages of 40–49, compare systolic blood pressure across race as reported in the `Race1` variable. Order the resulting table from lowest to highest average systolic blood pressure.

6

Importing data

We have been using datasets already stored as R objects. In data analysis work we rarely have such luck and will have to import data into R from either a file, a database, or another source. Currently, one of the most common ways of storing and sharing data for analysis is through electronic spreadsheets. A spreadsheet stores data in rows and columns. It is basically a file version of a data frame. When saving such a table to a computer file, one needs a way to define when a new row or column ends and the other begins. This in turn defines the cells in which single values are stored. Here is an example of what a comma separated file looks like if we open it with a basic text editor:



In this chapter, we outline how to load data from a file into R. First, it's crucial to identify the file's location; thus, we touch on file paths and working directories (detailed in Chapter 18). Next, we delve into file types (text or binary) and encodings (like ASCII and Unicode), both essential for data import. We then introduce popular functions for data importing, referred to as *parsers*. Lastly, we offer tips on how to store data in spreadsheets. Advanced topics like extracting data from websites or PDFs will be discussed in the book's Data Wrangling section.

DOI: 10.1201/9781003220923-6

6.1 Navigating and managing the filesystem

The first step when importing data from a spreadsheet is to locate the file containing the data. Although we do not recommend it, you can use an approach similar to what you do to open files in Microsoft Excel by clicking on the RStudio "File" menu, clicking "Import Dataset", then clicking through folders until you find the file. However, we write code rather than use the point-and-click approach. The key concepts we need to learn to do this are described in detail in Chapter 18. Here we provide an overview of the very basics.

6.1.1 The filesystem

You can think of your computer's filesystem as a series of nested folders, each containing other folders and files. We refer to folders as *directories*. We refer to the folder that contains all other folders as the *root directory*. We refer to the directory in which we are currently located as the *working directory*. The working directory therefore changes as you move through folders: think of it as your current location.

6.1.2 Relative and full paths

The *path* of a file is a list of directory names that can be thought of as instructions on what folders to click on, and in what order, to find the file. If these instructions are for finding the file from the root directory, we refer to it as the *full path*. If the instructions are for finding the file starting in the working directory, we refer to it as a *relative path*. Section 18.3 provides more details on this topic.

To see an example of a full path on your system type the following:

```
system.file(package = "dslabs")
#> [1] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/
        library/dslabs"
```

Note that the output will be different across different computers. The `system.file` function finds the full path to the files that were added to your system when you installed the **dslabs** package. The strings separated by slashes are the directory names. The first slash represents the root directory and we know this is a full path because it starts with a slash.

We can use the function `list.files` to show the names of files and directories in any directory. For example, here are the files in the **dslabs** package directory:

```
dir <- system.file(package = "dslabs")
list.files(dir)
#>  [1] "data"        "DESCRIPTION" "extdata"     "help"
#>  [5] "html"        "INDEX"       "Meta"        "NAMESPACE"
#>  [9] "R"           "script"
```

Note that these do not start with slash which implies they are *relative paths*. These relative paths give us the location of the files or directories if the path stored in `dir` is our working directory.

> **Note**
>
> You will not make much use of the `system.file` function in your day-to-day data analysis work. We introduce it in this section because it facilitates the sharing of spreadsheets that can be used to practice. The spreadsheets are in the `extdata` directory.

6.1.3 The working directory

We highly recommend only using relative paths in your code. The reason is that full paths are unique to your computer and you want your code to be portable. If you want to know the full path of your working directory using the `getwd` function. If you need to change your working directory, you can use the function `setwd` or you can change it through RStudio by clicking on "Session".

When you start a project you want to pick a directory to store all the files related to that project and make this is your working directory when running your analysis. This will facilitate th because if you provide a relative path to an importing functions, it will assume you want R to search for this file in the working directory. Chapter 20 provides details on how to organize projects with RStudio.

6.1.4 Generating path names

The `file.path` function combines characters to form a complete path, ensuring compatibility with the respective operating system. Linux and Mac use forward slashes `/`, while Windows uses backslashes \, to separate directories. This function is useful because often you want to define paths using a variable. Here is an example that constructs the full path for a spreadsheet containing the murders data. Here the variable `dir` contains the full path for the **dslabs** package and `extdata/murders.csv` is the relative path of the spreadsheet if `dir` is considered the working directory.

```
dir <- system.file(package = "dslabs")
file_path <- file.path(dir, "extdata/murders.csv")
```

You can copy the file with full path `file_path` to your working directory using the function `file.copy`:

```
file.copy(file_path, "murders.csv")
#> [1] TRUE
```

If the file is copied successfully, this function will return `TRUE`. Note that we used the same filename for the destination file, but we can give it whatever name we want. If a file with that name already exists in your destination directory, the copy will be unsuccessful. You can change this behavior with the `overwrite` argument.

ISTUDY

6.2 File types

For most data analysis applications, files can generally be classified into two categories: text files and binary files. In this section we describe the most widely used format for both these types and the best way to identify them. In the last subsection we describe the importance of knowing the file encoding.

> **Note**
>
> For this and the following section we assume you have copied the `murders.csv` file into your working directory. You can use the code at the end of the previous section to do this.

6.2.1 Text files

You have already worked with text files. All your R scripts and Quarto files, for example, are text files and so are the Quarto files used to create this book. The `murders.csv` file mentioned above is also text files. One big advantage of these files is that we can easily "look" at them without having to purchase any kind of special software or follow complicated instructions. Any text editor can be used to examine a text file, including freely available editors such as RStudio or nano. To see this, try opening a csv file using the "Open file" RStudio tool. You should be able to see the content right on your editor.

When text files are used to store a spreadsheet, line breaks are used to separate rows and a predefined character, referred to as the *delimiter*, is used to separate columns within a row. The most common delimiters are comma (`,`), semicolon (`;`), space (), and tab (a preset number of spaces or `\t`). Slightly different approaches are used to read these files into R, so we need to know what delimiter was used. In some cases, the delimiter can be inferred from file suffix. For example, files ending in `csv` or `tsv` are expected to be comma and tab delimited, respectively. However, it is harder to infer the delimiter for files ending in `txt`. As a result we recommend looking at the file rather than inferring from the suffix. You can look at any number of lines from within R using the `readLines` function:

```
readLines("murders.csv", n = 3)
#> [1] "state,abb,region,population,total"
#> [2] "Alabama,AL,South,4779736,135"
#> [3] "Alaska,AK,West,710231,19"
```

This immediately reveals that the file is indeed comma delimited. It also reveals that the file has a header: the first row contains column names rather than data. This is also important to know. Most parsers assume the file starts with a header, but not all files have one.

6.2.2 Binary files

Opening image files such as jpg or png in a text editor or using `readLines` in R will not show comprehensible content because these are *binary* files. Unlike text files, which are designed for human readability and have standardized conventions, binary files can adopt numerous formats specific to their data type. While R's `readBin` function can process any binary file, interpreting the output necessitates a thorough understanding of the file's structure. This

ISTUDY

intricate topic isn't covered in this book. Instead, we concentrate on the prevalent binary formats for spreadsheets: Microsoft Excel's xls and xlsx.

6.2.3 Encoding

A frequent issue when importing data, whether text or binary, is incorrectly identifying the file's *encoding*. At its core, a computer translates everything into sequences of 0s and 1s. ASCII is an *encoding* system that assigns specific numbers to characters. Using 7 bits, ASCII can represent $2^7 = 128$ unique symbols, sufficient for all English keyboard characters. However, many global languages contain characters outside ASCII's range. For instance, the é in "México" isn't in ASCII's catalog. To address this, broader encodings, such as Unicode, emerged. Unicode offers variations using 8, 16, or 32 bits, known as UTF-8, UTF-16, and UTF-32. RStudio typically uses UTF-8 as its default. Notably, ASCII is a subset of UTF-8, meaning that if a file is ASCII-encoded, presuming it's UTF-8 encoded won't cause issues. However, there other encodings, such as ISO-8859-1 (also known as Latin-1) developed for the western European languages, Big5 for Traditional Chinese, and ISO-8859-6 for Arabic.

The **dslabs** package includes a file that is not UTF-8 encoded to serve as an example. Notice the strange characters that appear you attempt to read in the first line:

```
fn <- "calificaciones.csv"
file.copy(file.path(system.file("extdata", package = "dslabs"), fn), fn)
#> [1] TRUE
readLines(fn, n = 1)
#> [1] "\"nombre\",\"f.n.\",\"estampa\",\"puntuaci\xf3n\""
```

In the following section, we'll introduce several helpful import functions, some of which allow you to specify the file encoding.

6.3 Parsers

Importing functions, or parsers, are available from base R. However, more powerful and often faster functions are available in the **readr**, **readxl**, and **data.table** packages. In this section we review some examples. We also describe how data can be downloaded or read directly from the internet.

6.3.1 Base R

Base R provides several file parsers for example `read.csv`, `read.table`, and `read.delim`. The first argument can take either a full or relative path. If a relative path is provided, the parser assumes you want to search in the working directory. Therefore, to read the `murders.csv` file previously copied to our working directory, we can simply type:

```
dat <- read.csv("murders.csv")
```

An often useful R-base importing function is `scan`, as it provides much flexibility. When reading in spreadsheets many things can go wrong. The file might have multiline headers or be missing cells. With experience you will learn how to deal with different challenges.

ISTUDY

Carefully reading the help files for the functions discussed here will be useful. With scan you can read-in each cell of a file. Here is an example:

```
x <- scan("murders.csv", sep = ",", what = "c")
x[1:10]
#>  [1] "state"      "abb"        "region"     "population" "total"
#>  [6] "Alabama"    "AL"         "South"      "4779736"    "135"
```

6.3.2 readr

The **readr** package includes parsers, for reading text file spreadsheets into R. **readr** is part of the **tidyverse**, but you can load it directly using:

```
library(readr)
```

The following functions are available to read-in text file spreadsheets:

Function	Format	Typical suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must define delimiter	txt

It also includes `read_lines` with similar functionality to `readLines`.

We can read in the `murders.csv` file using

```
dat <- read_csv("murders.csv")
#> Rows: 51 Columns: 5
#> -- Column specification --------------------------------------------------
#> Delimiter: ","
#> chr (3): state, abb, region
#> dbl (2): population, total
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet
#> this message.
```

Note that we receive a message letting us know what data types were used for each column. Also note that `dat` is a `tibble`, not just a data frame. We can suppress this message using the argument `show_col_types = FALSE`.

The **readr** parsers permit us to specify an encoding. It also includes a function that tries to guess the encoding:

ISTUDY

```
guess_encoding("murders.csv")
#> # A tibble: 1 x 2
#>   encoding confidence
#>   <chr>         <dbl>
#> 1 ASCII             1
```

This function can help us read the file we previously noted was showing strange characters:

```
guess_encoding("calificaciones.csv")
#> # A tibble: 3 x 2
#>   encoding   confidence
#>   <chr>           <dbl>
#> 1 ISO-8859-1       0.92
#> 2 ISO-8859-2       0.72
#> 3 ISO-8859-9       0.53
```

Once we know the encoding we can specify it through the `locale` argument:

```
dat <- read_csv("calificaciones.csv", show_col_types = FALSE,
                locale = locale(encoding = "ISO-8859-1"))
```

We learn about locales in Chapter 14.

We can now see that the characters in the header were read in correctly:

```
names(dat)
#> [1] "nombre"     "f.n."       "estampa"    "puntuación"
```

6.3.3 readxl

The **readxl** package provides functions to read-in Microsoft Excel formats.

```
library(readxl)
```

The main functions are:

Function	Format	Typical suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

The Microsoft Excel formats permit you to have more than one spreadsheet in one file. These are referred to as *sheets*. The functions listed above read the first sheet by default, but we can also read the others. The `excel_sheets` function gives us the names of all the sheets in an Excel file. These names can then be passed to the `sheet` argument in the three functions above to read sheets other than the first.

ISTUDY

6.3.4 data.table

The **data.table** package provides the `fread` function, a powerful and fast utility designed for reading large datasets. `fread` automatically detects the format of the input, whether it's delimited text or even files compressed in formats like gzip or zip. It offers a significant speed advantage over the other parsers described here, especially for large files.

```
library(data.table)
dat <- fread("murders.csv")
```

Note `fread` returns a `data.table` object.

6.3.5 Downloading files

A common place for data to reside is on the internet. When these data are in files, we can download them and then import them or even read them directly from the web. For example, we note that because our **dslabs** package is on GitHub, the file we downloaded with the package has a url:

```
url <- paste0("https://raw.githubusercontent.com/",
              "rafalab/dslabs/master/inst/extdata/murders.csv")
```

Most parsers can read these files directly:

```
dat <- read.csv(url)
```

If you want to have a local copy of the file, you can use the `download.file` function:

```
download.file(url, "murders.csv")
```

This will download the file and save it on your system with the name `murders.csv`. You can use any name here, not necessarily `murders.csv`.

> **Warning**
>
> The function `download.file` overwrites existing files without warning.

Two functions that are sometimes useful when downloading data from the internet are `tempdir` and `tempfile`. The first creates a directory with a random name that is very likely to be unique. Similarly, `tempfile` creates a character string, not a file, that is likely to be a unique filename. So you can run a command like this which erases the temporary file once it imports the data:

```
tmp_filename <- tempfile()
download.file(url, tmp_filename)
dat <- read_csv(tmp_filename)
file.remove(tmp_filename)
```

ISTUDY

6.4 Organizing data with spreadsheets

Although this book focuses almost exclusively on data analysis, data management is also an important part of data science operations. As explained in the introduction, we do not cover this topic. However, quite often data analysts need to collect data, or work with others collecting data, in a way that is most conveniently stored in a spreadsheet. Although filling out a spreadsheet by hand is a practice we highly discourage, and we instead recommend the process be automatized as much as possible, sometimes you just have to do it. Therefore, in this section, we provide recommendations on how to organize data in a spreadsheet. Although there are R packages designed to read Microsoft Excel spreadsheets, we generally want to avoid this format. Instead, we recommend Google Sheets as a free software tool. Below we summarize the recommendations made in paper by Karl Broman and Kara Woo[1]. Please read the paper for important details.

- **Be Consistent** – Before you commence entering data, have a plan. Once you have a plan, be consistent and stick to it.
- **Choose Good Names for Things** – You want the names you pick for objects, files, and directories to be memorable, easy to spell, and descriptive. This is actually a hard balance to achieve and it does require time and thought. One important rule to follow is **do not use spaces**, use underscores _ or dashes instead. Also, avoid symbols; stick to letters and numbers.
- **Write Dates as YYYY-MM-DD** – To avoid confusion, we strongly recommend using this global ISO 8601 standard.
- **No Empty Cells** – Fill in all cells and use some common code for missing data.
- **Put Just One Thing in a Cell** – It is better to add columns to store the extra information rather than having more than one piece of information in one cell.
- **Make It a Rectangle** – The spreadsheet should be a rectangle.
- **Create a Data Dictionary** – If you need to explain things, such as what the columns are or what the labels used for categorical variables are, do this in a separate file.
- **No Calculations in the Raw Data Files** – Excel permits you to perform calculations. Do not make this part of your spreadsheet. Code for calculations should be in a script.
- **Do Not Use Font Color or Highlighting as Data** – Most import functions are not able to import this information. Encode this information as a variable instead.
- **Make Backups** – Make regular backups of your data.
- **Use Data Validation to Avoid Errors** – Leverage the tools in your spreadsheet software so that the process is as error-free and repetitive-stress-injury-free as possible.
- **Save the Data as Text Files** – Save files for sharing in comma or tab delimited format.

6.5 Exercises

1. Use the `read_csv` function to read each of the files that the following code saves in the `files` object:

[1] https://www.tandfonline.com/doi/abs/10.1080/00031305.2017.1375989

ISTUDY

```
path <- system.file("extdata", package = "dslabs")
files <- list.files(path)
files
```

2. Note that the the `olive` file, gives us a warning. This is because the first line of the file is missing the header for the first column.

   Read the help file for `read_csv` to figure out how to read in the file without reading this header. If you skip the header, you should not get this warning. Save the result to an object called `dat`.

3. A problem with the previous approach is that we don't know what the columns represent. Type:

   ```
   names(dat)
   ```

   to see that the names are not informative.

   Use the `readLines` function to read in just the first line.

4. Pick a measurement you can take on a regular basis. For example, your daily weight or how long it takes you to run 5 miles. Keep a spreadsheet that includes the date, the hour, the measurement, and any other informative variable you think is worth keeping. Do this for 2 weeks. Then make a plot.

ISTUDY

Part II

Data Visualization

ISTUDY



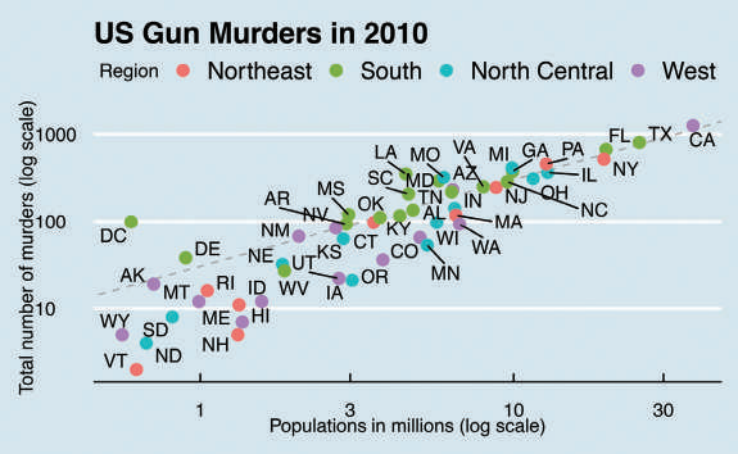
Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

ISTUDY

Looking at the numbers and character strings that define a dataset is rarely useful. To convince yourself, print and stare at the US murders data table:

```
library(dslabs)
head(murders)
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

What do you learn from staring at this table? How quickly can you determine which states have the largest populations? Which states have the smallest? How large is a typical state? Is there a relationship between population size and total murders? How do murder rates vary across regions of the country? For most human brains, it is quite difficult to extract this information just by looking at the numbers. In contrast, the answer to all the questions above are readily available from examining this plot:



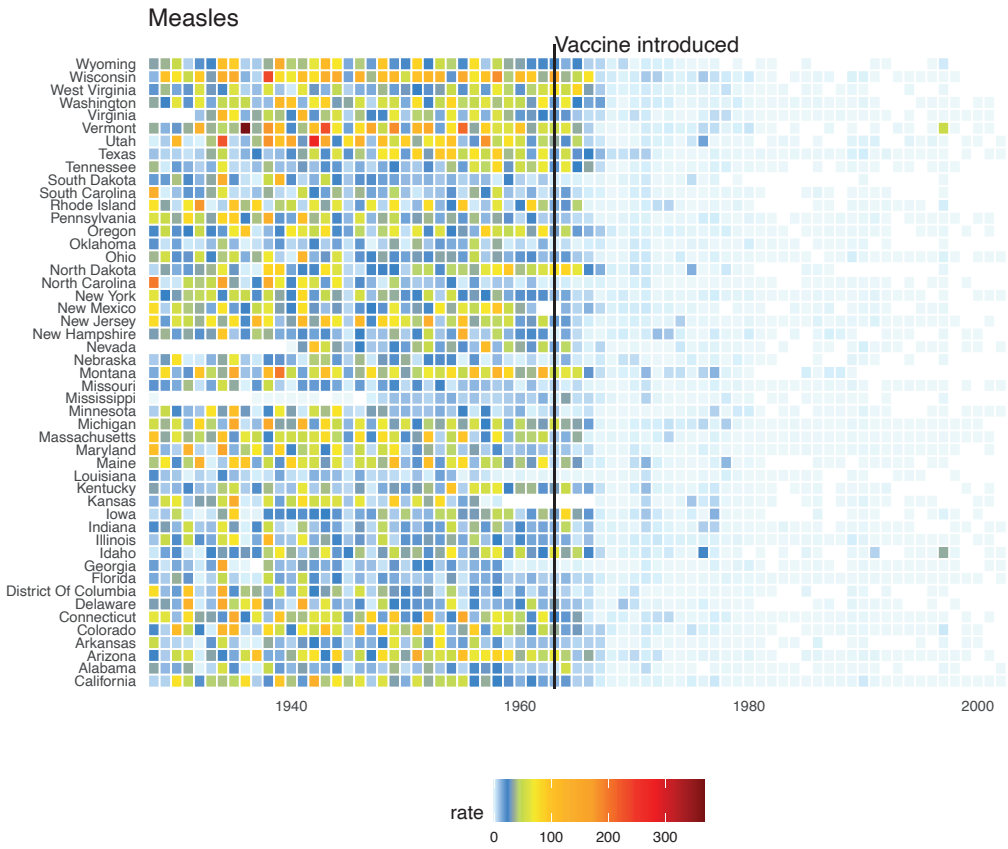
We are reminded of the saying "a picture is worth a thousand words". Data visualization provides a powerful way to communicate a data-driven finding. In some cases, the visualization is so convincing that no follow-up analysis is required.

The growing availability of informative datasets and software tools has led to increased reliance on data visualizations across many industries, academia, and government. A salient example is news organizations, which are increasingly embracing *data journalism* and including effective *infographics* as part of their reporting.

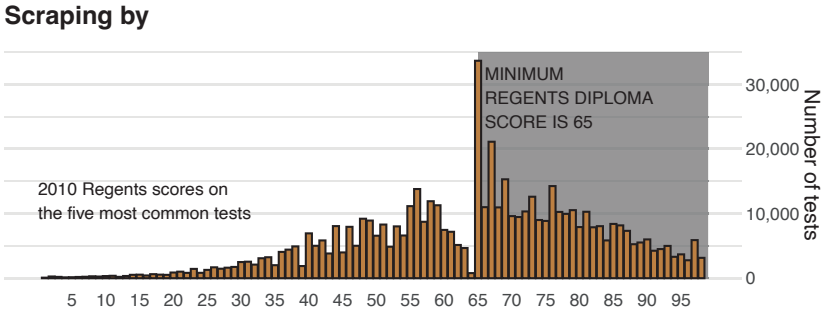
A particularly effective example is a Wall Street Journal article[2] showing data related to the impact of vaccines on battling infectious diseases. One of the graphs shows measles cases by US state through the years with a vertical line demonstrating when the vaccine was introduced.

[2] http://graphics.wsj.com/infectious-diseases-and-vaccines/?mc_cid=711ddeb86e

ISTUDY



A New York Times chart provides a compelling example by summarizing scores from the NYC Regents Exams[3]. According to the accompanying article[4], these scores are collected for various purposes, one of which is to determine a student's eligibility for high school graduation. In New York City, a score of 65 is required to pass. The pattern of these test scores suggests something unusual.



[3] http://graphics8.nytimes.com/images/2011/02/19/nyregion/19schoolsch/19schoolsch-popup.gif
[4] https://www.nytimes.com/2011/02/19/nyregion/19schools.html

ISTUDY

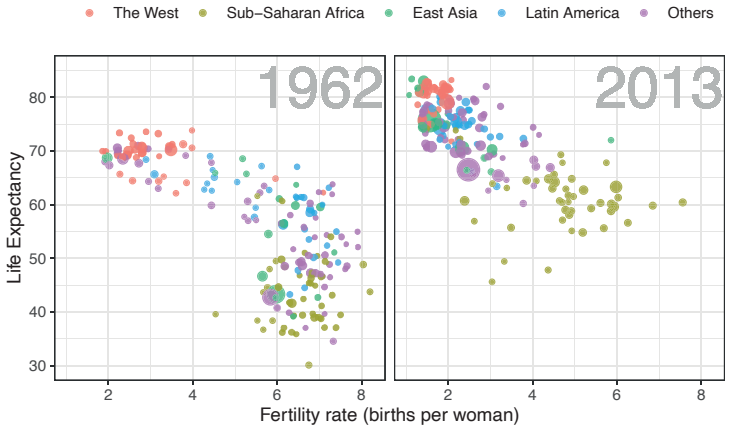
The most common test score is the minimum passing grade, with very few scores just below the threshold. This unexpected result is consistent with students close to passing having their scores bumped up.

This is an example of how data visualization can lead to discoveries which would otherwise be missed if we simply subjected the data to a battery of data analysis tools or procedures. Data visualization is the strongest tool of what we call *exploratory data analysis* (EDA). John W. Tukey[5], considered the father of EDA, once said,

> "The greatest value of a picture is when it forces us to notice what we never expected to see."

Many widely used data analysis tools were initiated by discoveries made via EDA. EDA is perhaps the most important part of data analysis, yet it is one that is often overlooked.

Data visualization is also now pervasive in philanthropic and educational organizations. In the talks New Insights on Poverty[6] and The Best Stats You've Ever Seen[7], Hans Rosling forces us to notice the unexpected with a series of plots related to world health and economics. In his videos, he uses animated graphs to show us how the world is changing and how old narratives are no longer true.



It is also important to note that mistakes, biases, systematic errors and other unexpected problems often lead to data that should be handled with care. Failure to discover these problems can give rise to flawed analyses and false discoveries. As an example, consider that measurement devices sometimes fail and that most data analysis procedures are not designed to detect these. Yet these data analysis procedures will still give you an answer. The fact that it can be difficult or impossible to notice an error just from the reported results makes data visualization particularly important.

In this part of the book, we will learn the basics of data visualization and exploratory data analysis by using three motivating examples. We will use the **ggplot2** package to code. To learn the very basics, we will start with a somewhat artificial example: heights reported by

[5] https://en.wikipedia.org/wiki/John_Tukey
[6] https://www.ted.com/talks/hans_rosling_reveals_new_insights_on_poverty?language=en
[7] https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen

ISTUDY

students. Then we will focus on two cases studies: 1) world health and economics and 2) infectious disease trends in the United States. Note that we do not cover interactive graphics. For those interested in creating interactive plots we highly recommend learning to use the **plotly** package or Shiny[8], both build on top of R. For more advanced challenges consider learning to program in D3.js[9].

[8] https://shiny.rstudio.com/
[9] https://d3js.org/

ISTUDY

7

Visualizing data distributions

Summarizing complex datasets is crucial in data analysis, allowing us to share insights drawn from the data more effectively. One common method is to use the *average* value to summarize a list of numbers. For instance, a high school's quality might be represented by the average score in a standardized test. Sometimes, an additional value, the *standard deviation*, is added. So, a report might say the scores were 680 ± 50, boiling down a full set of scores to just two numbers. But is this enough? Are we overlooking crucial information by relying solely on these summaries instead of the complete data?

Our first data visualization building block is learning to summarize lists of numbers or categories. More often than not, the best way to share or explore these summaries is through data visualization. The most basic statistical summary of a list of objects or numbers is its distribution. Once a data has been summarized as a distribution, there are several data visualization techniques to effectively relay this information. For this reason, it is important to have a deep understand of the concept of a distribution.

In this chapter, we discuss properties of a variety of distributions and how to visualize distributions using a motivating example of student heights.

7.1 Variable types

The two main variables types are *categorical* and *numeric*. Each can be divided into two other groups: categorical can be ordinal or not, whereas numerical variables can be discrete or continuous. When each entry in a dataset comes from one of a small number of groups, we refer to the data as *categorical data*. Two simple examples are sex (male or female) and US regions (Northeast, South, North Central, and West). Some categorical data can be ordered even if they are not numbers per se, such as spiciness (mild, medium, and hot). In statistics, ordered categorical data are referred to as *ordinal* data. Examples of numerical data are population sizes, murder rates, and heights. We can further divide numerical data into continuous and discrete. Continuous variables are those that can take any value, such as heights, if measured with enough precision. For example, a pair of twins may be 68.12 and 68.11 inches, respectively. Counts, such as population sizes, are discrete because they have to be round numbers.

Keep in mind that discrete numeric data can be considered ordinal. Although this is technically true, we usually reserve the term ordinal data for variables belonging to a small number of different groups, with each group having many members. In contrast, when we have many groups with few cases in each group, we typically refer to them as discrete numerical variables. So, for example, the number of packs of cigarettes a person smokes a day, rounded to the closest pack, would be considered ordinal, while the actual number of

DOI: 10.1201/9781003220923-7

cigarettes would be considered a numerical variable. But, indeed, there are examples that can be considered both numerical and ordinal when it comes to visualizing data.

Here we focus on numeric variables because visualizing this data type is substantially more complex. However, we start by describing data visualization and summarization approaches for categorical data.

7.2 Case study: describing student heights

We introduce a new motivating problem. It is an artificial one, but it will help us illustrate the concepts needed to understand distributions.

Pretend that we have to describe the heights of our classmates to ET, an extraterrestrial that has never seen humans. As a first step, we need to collect data. To do this, we ask students to report their heights in inches. We ask them to provide sex information because we know there are two different height distributions by sex. We collect the data and save it in the `heights` data frame:

```
library(tidyverse)
library(dslabs)
head(heights)
#>      sex height
#> 1   Male     75
#> 2   Male     70
#> 3   Male     68
#> 4   Male     74
#> 5   Male     61
#> 6 Female     65
```

One way to convey the heights to ET is to simply send him this list of 1,050 heights. But there are much more effective ways to convey this information, and understanding the concept of a distribution will be key. To simplify the explanation, we first focus on male heights. We examine the female height data in Section 7.6.

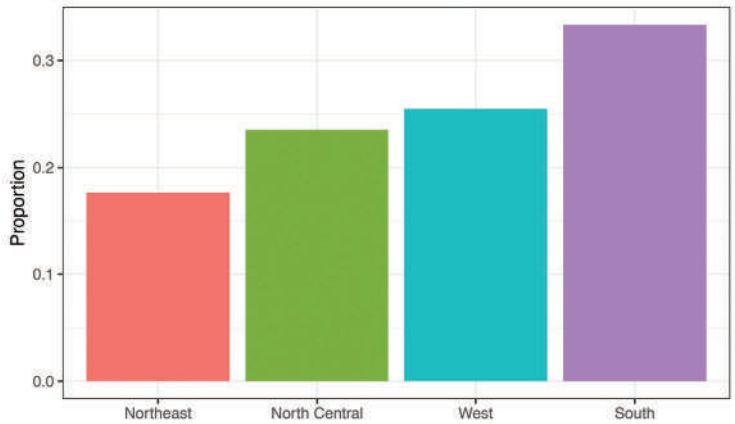
7.3 Distributions

The most basic statistical summary of a list of objects or numbers is its *distribution*. The simplest way to think of a distribution is as a compact description of a list with many entries. This concept should not be new for readers of this book. For example, with categorical data, the distribution simply describes the proportion of each unique category. The sex represented in the heights dataset is:

ISTUDY

```
#>
#> Female   Male
#>  0.227  0.773
```

This two-category *frequency table* is the simplest form of a distribution. We don't really need to visualize it since one number describes everything we need to know: 23% are females and the rest are males. When there are more categories, then a simple barplot describes the distribution. Here is an example with US state regions:



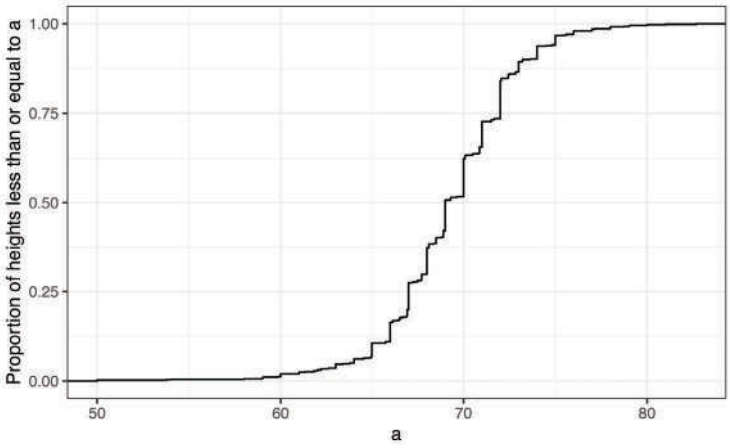
This particular plot simply shows us four numbers, one for each category. We usually use barplots to display a few numbers. Although this particular plot does not provide much more insight than a frequency table itself, it is a first example of how we convert a vector into a plot that succinctly summarizes all the information in the vector. When the data is numerical, the task of displaying distributions is more challenging.

7.3.1 Histograms

Numerical data that are not categorical also have distributions. However, in general, when data is not categorical, reporting the frequency of each entry, as we did for categorical data, is not an effective summary since most entries are unique. For example, in our case study, while several students reported a height of 68 inches, only one student reported a height of `68.503937007874` inches and only one student reported a height `68.8976377952756` inches. We assume that they converted from 174 and 175 centimeters, respectively.

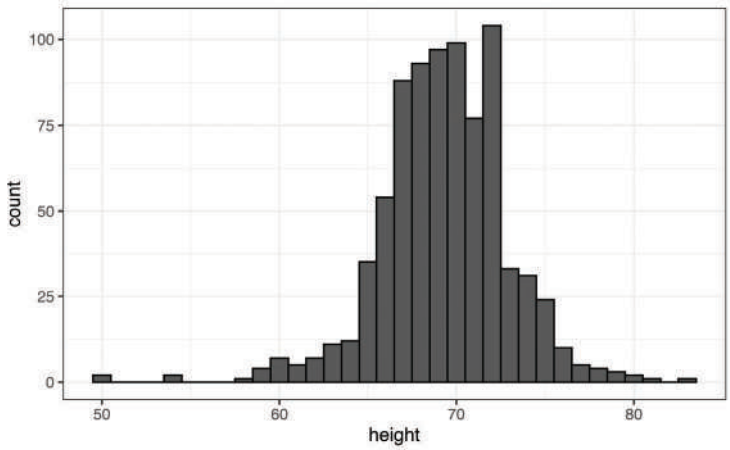
Statistics textbooks teach us that a more useful way to define a distribution for numeric data is to define a function that reports the proportion of the data below a for all possible values of a. This function is called the empirical cumulative distribution function (eCDF), it can be plotted, and it provides a full description of the distribution of our data. Here is the eCDF for male student heights:

ISTUDY



However, summarizing data by plotting the eCDF is actually not very popular in practice. The main reason is that it does not easily convey characteristics of interest such as: at what value is the distribution centered? Is the distribution symmetric? What ranges contain 95% of the values?

Histograms are much preferred because they greatly facilitate answering such questions. Histograms sacrifice just a bit of information to produce plots that are much easier to interpret. The simplest way to make a histogram is to divide the span of our data into non-overlapping bins of the same size. Then, for each bin, we count the number of values that fall in that interval. The histogram plots these counts as bars with the base of the bar defined by the intervals. Here is the histogram for the height data splitting the range of values into one inch intervals: $(49.5, 50.5]$, $(50.5, 51.5]$, $(51.5, 52.5]$, $(52.5, 53.5]$,..., $(82.5, 83.5]$.



As you can see in the figure above, a histogram is similar to a barplot, but it differs in that the x-axis is numerical, not categorical.

ISTUDY

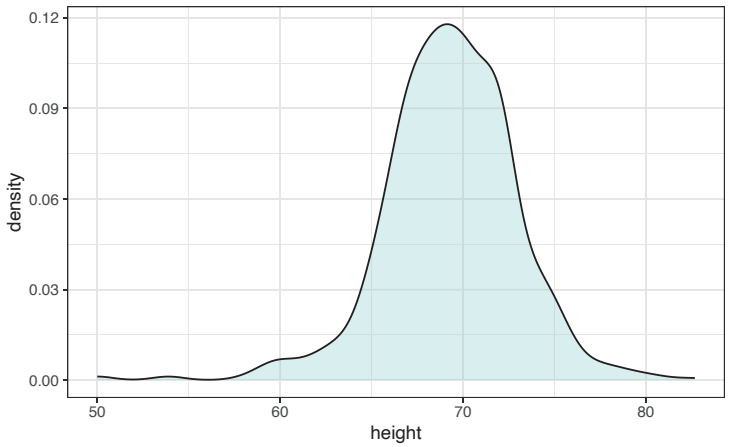
If we send this plot to ET, he will immediately learn some important properties about our data. First, the range of the data is from 50 to 84 with the majority (more than 95%) between 63 and 75 inches. Second, the heights are close to symmetric around 69 inches. Also, by adding up counts, ET could obtain a very good approximation of the proportion of the data in any interval. Therefore, the histogram above is not only easy to interpret, but also provides almost all the information contained in the raw list of 812 male heights with about 30 bins.

What information do we lose? Note that all values in each interval are treated the same when computing bin heights. So, for example, the histogram does not distinguish between 64, 64.1, and 64.2 inches. Given that these differences are almost unnoticeable to the eye, the practical implications are negligible and we were able to summarize the data to just 23 numbers.

We discuss how to code histograms in Section 8.14.

7.3.2 Smoothed density

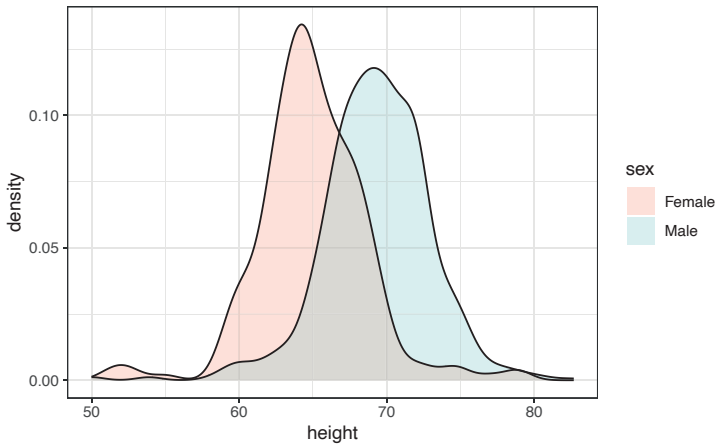
Smooth density plots relay the same information as a histogram but are aesthetically more appealing. Here is what a smooth density plot looks like for our male heights data:



In this plot, we no longer have sharp edges at the interval boundaries and many of the local peaks have been removed. Also, the scale of the y-axis changed from counts to *density*. To fully understand smooth densities, we have to understand *estimates*, a topic covered in statistics our advanced data science textbooks. Here we simply describe them as making the histograms prettier by drawing a curve that goes through the top of the histogram bars and then removing the bars. The values shown y-axis are chosen so that the area under the curve adds up to 1. This implies that for any interval, the area under the curve for that interval gives us an approximation of how what proportion of the data is in the interval.

An advantage of smooth densities over histograms for visualization purposes is that densities make it easier to compare two distributions. This is in large part because the jagged edges of the histogram add clutter. Here is an example comparing male and female heights:

ISTUDY

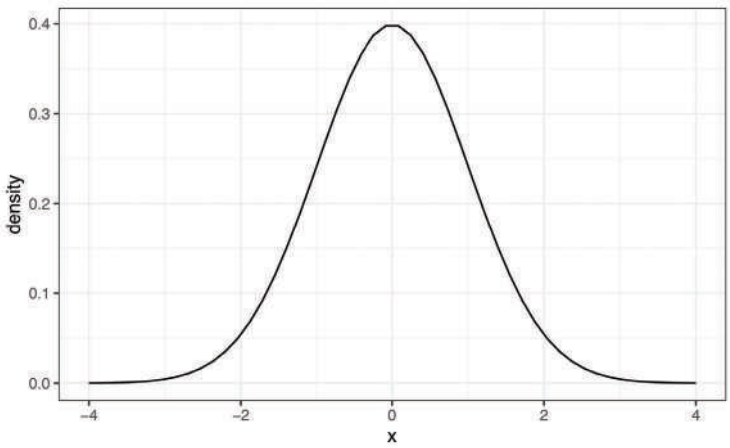


With the right argument, `ggplot` automatically shades the intersecting region with a different color. We will show examples of **ggplot2** code for densities in Chapter 10 as well as Section 8.14.

7.3.3 The normal distribution

Histograms and density plots provide excellent summaries of a distribution. But can we summarize even further? We often see the average and standard deviation used as summary statistics: a two-number summary! To understand what these summaries are and why they are so widely used, we need to understand the normal distribution.

The normal distribution, also known as the bell curve and as the Gaussian distribution. Here is what the normal distribution looks like:



ISTUDY

The normal distribution is one of the most famous mathematical concepts in history. A reason for this is that the distribution of many datasets can be approximated with normal distributions. These include gambling winnings, heights, weights, blood pressure, standardized test scores, and experimental measurement errors. Statistical textbooks offer explanations for why this is the case. But how can the same distribution approximate datasets with completely different ranges for values, for example heights and weights? A second important characteristic of the normal distribution is that it can be adapted to different datasets by just adjusting two numbers, referred to as the *average* or *mean* and the *standard deviation* (SD). The normal distribution is symmetric, centered at what we refer to as the average, and most values (about 95%) are within 2 SDs from the average. The plot above shows a normal distribution with average 0 and SD 1, often referred to as a *standard normal*. Note that the fact that only two numbers are needed to adapt the normal distribution to a dataset implies that if our data distribution is approximated by a normal distribution, all the information needed to describe the distribution can be encoded by just two numbers. We now define these values for an arbitrary list of numbers.

Once we are convinced that our data, say it is stored in the vector `x`, has a distribution that is *approximately normal*, we can find the specific one that matches our data by matching the average and SD of the data to the average and SD of the normal distribution, respectively. For a list of numbers contained in a vector `x`:

```
index <- heights$sex == "Male"
x <- heights$height[index]
```

the average is defined as

```
m <- sum(x) / length(x)
```

and the SD is defined as

```
s <- sqrt(sum((x - mu)^2) / length(x))
```

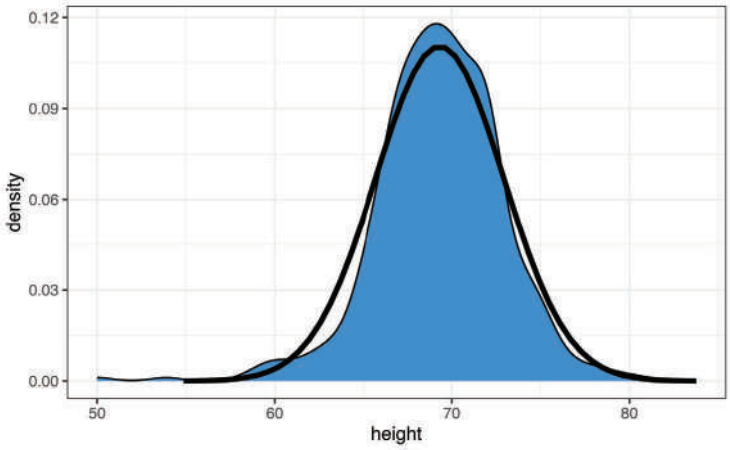
which can be interpreted as the average distance between values and their average.

The pre-built functions `mean` and `sd` (note that, for reasons explained in statistics textbooks,`sd` divides by `length(x)-1` rather than `length(x)`) can be used here:

```
m <- mean(x)
s <- sd(x)
c(average = m, sd = s)
#> average      sd
#>   69.31    3.61
```

Here is a plot of our student height smooth density in blue and the normal distribution with mean = 69.3 and SD = 3.6 plotted as a black line:

ISTUDY

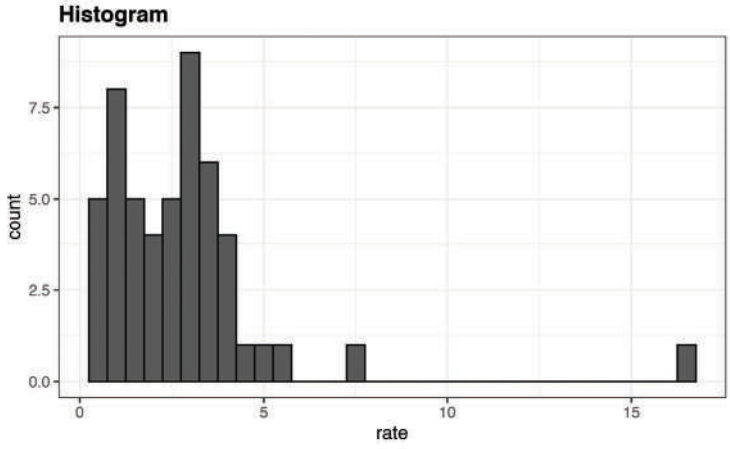


7.4 Boxplots

To understand boxplots we need to define some terms that are commonly used in exploratory data analysis.

The *percentiles* are the values for which $p = 0.01, 0.02, ..., 0.99$ of the data are less then or equal to that value, respectively. We call, for example, the case of $p = 0.10$ the 10th percentile, which gives us a number for which 10% of the data is below. The most famous percentile is the 50th, also known as the *median*. Another special case that receives a name are the *quartiles*, which are obtained when setting $p = 0.25, 0.50$, and 0.75, which are used by the boxplot.

To motivate boxplots we will go back to the US murder data. Suppose we want to summarize the murder rate distribution. Using the data visualization technique we have learned, we can quickly see that the normal approximation does not apply here:

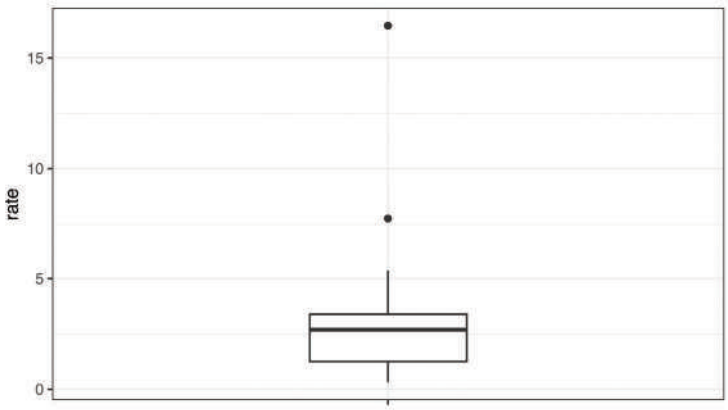


In this case, the histogram above or a smooth density plot would serve as a relatively succinct summary.

ISTUDY

Now suppose those used to receiving just two numbers as summaries ask us for a more compact numerical summary.

The *boxplot* provides a five-number summary composed of the range (the minimum and maximum) along with the *quartiles* (the 25th, 50th, and 75th percentiles). The R implementation of boxplots ignore *outliers* when computing the range and instead plot these as independent points. The help file provides a specific definition of outliers. The boxplot shows these numbers as a "box" with "whiskers"



with the box defined by the 25% and 75% percentile and the whiskers showing the range. The distance between the 25% and 75% percentile is called the *interquartile* range. The two points are outliers according to the R implementation. The median is shown with a horizontal line.

From just this simple plot, we know that the median is about 2.5, that the distribution is not symmetric, and that the range is 0–5 for the great majority of states with two exceptions.

We discuss how to make boxplots in Section 8.14.

7.5 Stratification

In data analysis we often divide observations into groups based on the values of one or more variables associated with those observations. For example in the next section we divide the height values into groups based on a sex variable: females and males. We call this procedure *stratification* and refer to the resulting groups as *strata*.

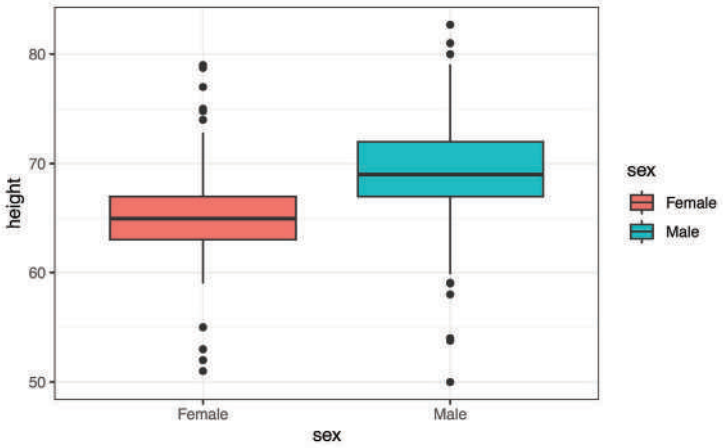
Stratification is common in data visualization because we are often interested in how the distributions of variables differ across different subgroups. We will see several examples throughout this part of the book, starting with the next section.

ISTUDY

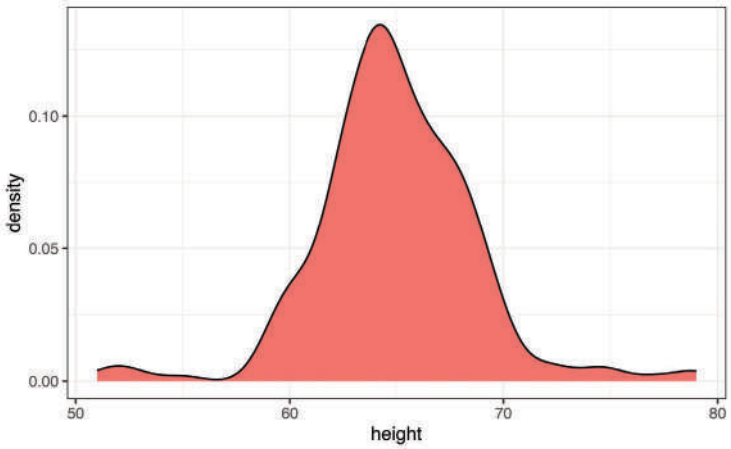
7.6 Case study continued

If we are convinced that the male height data is well approximated with a normal distribution we can report back to ET a very succinct summary: male heights follow a normal distribution with an average of 69.3 inches and a SD of 3.6 inches. With this information, ET will have a good idea of what to expect when he meets our male students. However, to provide a complete picture we need to also provide a summary of the female heights.

Boxplots are useful when we want to quickly compare two or more distributions. Here are the heights for men and women:



The plot immediately reveals that males are, on average, taller than females. The interquartile ranges appear to be similar. But does the normal approximation also work for the female height data collected by the survey? We expect that they will follow a normal distribution, just like males. However, exploratory plots reveal that the approximation is not as useful:



ISTUDY

We see something we did not see for the males: the density plot has a second "bump." Also, the highest points tend to be taller than expected by the normal than expected heights for a normal distribution. When reporting back to ET, we might need to provide a histogram rather than just the average and standard deviation for the female heights.

However, go back and read Tukey's quote. We have noticed what we didn't expect to see. If we look at other female height distributions, we do find that they are well approximated with a normal distribution. So why are our female students different? Is our class a requirement for the female basketball team? Are small proportions of females claiming to be taller than they are? Another, perhaps more likely, explanation is that in the form students used to enter their heights, `FEMALE` was the default sex and some males entered their heights, but forgot to change the sex variable. In any case, data visualization has helped discover a potential flaw in our data.

Regarding the five smallest values, note that these values are:

```
heights |> filter(sex == "Female") |>
  top_n(5, desc(height)) |>
  pull(height)
#> [1] 51 53 55 52 52
```

Because these are reported heights, a possibility is that the student meant to enter `5'1"`, `5'2"`, `5'3"`, or `5'5"`.

7.7 Exercises

1. Define variables containing the heights of males and females like this:

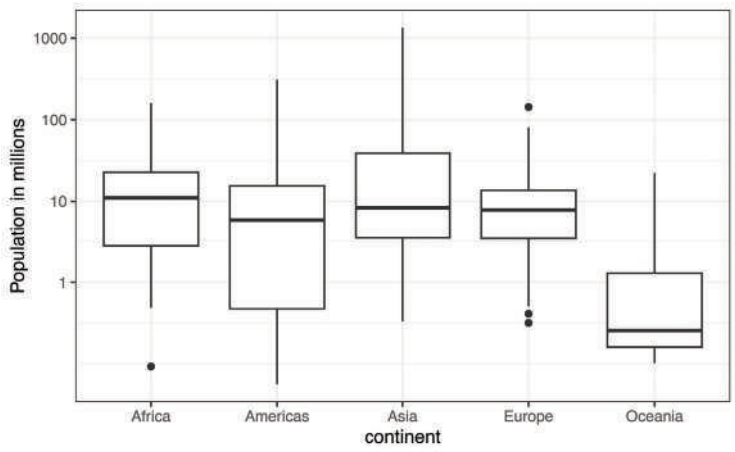
   ```
   library(dslabs)
   male <- heights$height[heights$sex == "Male"]
   female <- heights$height[heights$sex == "Female"]
   ```

   How many measurements do we have for each?

2. Suppose we can't make a plot and want to compare the distributions side by side. We can't just list all the numbers. Instead, we will look at the percentiles. Create a five row table showing `female_percentiles` and `male_percentiles` with the 10th, 30th, 50th, 70th, and 90th percentiles for each sex. Then create a data frame with these two as columns.

ISTUDY

3. Study the following boxplots showing population sizes by country:



   Which continent has the country with the biggest population size?

4. What continent has the largest median population size?

5. What is median population size for Africa to the nearest million?

6. What proportion of countries in Europe have populations below 14 million?

   (a) 0.99
   (b) 0.75
   (c) 0.50
   (d) 0.25

7. If we use a log transformation, which continent shown above has the largest interquartile range?

8. Load the height data set and create a vector `x` with just the male heights:

```
library(dslabs)
x <- heights$height[heights$sex=="Male"]
```

   What proportion of the data is between 69 and 72 inches (taller than 69, but shorter or equal to 72)? Hint: use a logical operator and `mean`.

ISTUDY

8

ggplot2

Exploratory data visualization is perhaps the greatest strength of R. One can quickly go from idea to data to plot with a unique balance of flexibility and ease. For example, Excel may be easier than R for some plots, but it is nowhere near as flexible. D3.js may be more flexible and powerful than R, but it takes much longer to generate a plot.

Throughout the book, we will be creating plots using the **ggplot2**[1] package.

```
library(dplyr)
library(ggplot2)
```

Many other approaches are available for creating plots in R. In fact, the plotting capabilities that come with a basic installation of R are already quite powerful. There are also other packages for creating graphics such as **grid** and **lattice**. We chose to use **ggplot2** in this book because it breaks plots into components in a way that permits beginners to create relatively complex and aesthetically pleasing plots using syntax that is intuitive and comparatively easy to remember.

One reason **ggplot2** is generally more intuitive for beginners is that it uses a grammar of graphics[2], the *gg* in **ggplot2**. This is analogous to the way learning grammar can help a beginner construct hundreds of different sentences by learning just a handful of verbs, nouns, and adjectives without having to memorize each specific sentence. Similarly, by learning a handful of **ggplot2** building blocks and its grammar, you will be able to create hundreds of different plots.

Another reason **ggplot2** is easy for beginners is that its default behavior is carefully chosen to satisfy the great majority of cases and is visually pleasing. As a result, it is possible to create informative and elegant graphs with relatively simple and readable code.

One limitation is that **ggplot2** is designed to work exclusively with data tables in tidy format. However, a substantial percentage of datasets that beginners work with are in, or can be converted into, this format. An advantage of this approach is that, assuming that our data is tidy, **ggplot2** simplifies plotting code and the learning of grammar for a variety of plots.

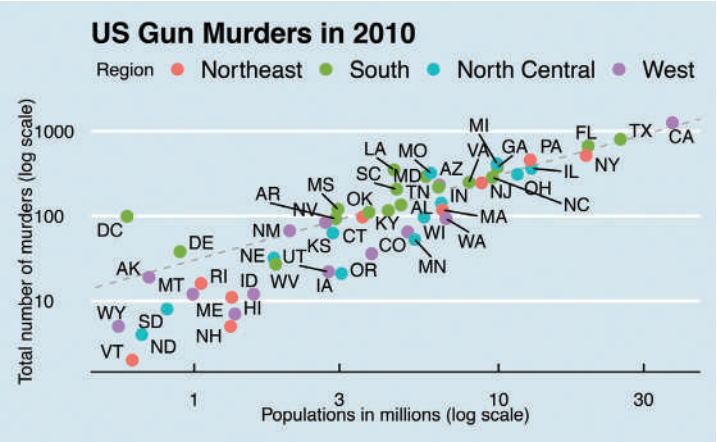
To use **ggplot2** you will have to learn several functions and arguments. These are hard to memorize, so we highly recommend you have the ggplot2 cheat sheet handy. You can get a copy from Posit's website[3] or simply perform an internet search for "ggplot2 cheat sheet."

[1] https://ggplot2.tidyverse.org/
[2] http://www.springer.com/us/book/9780387245447
[3] https://posit.co/resources/cheatsheets/

DOI: 10.1201/9781003220923-8

8.1 The components of a graph

We will construct a graph that summarizes the US murders dataset that looks like this:



We can clearly see how much states vary across population size and the total number of murders. Not surprisingly, we also see a clear relationship between murder totals and population size. A state falling on the dashed gray line has the same murder rate as the US average. The four geographic regions are denoted with color, which depicts how most southern states have murder rates above the average.

This data visualization shows us pretty much all the information in the data frame The code needed to make this plot is relatively simple. We will learn to create the plot part by part.

The first step in learning **ggplot2** is to be able to break a graph apart into components. Let's break down the plot above and introduce some of the **ggplot2** terminology. The main three components to note are:

- **Data**: The US murders data frame is being summarized. We refer to this as the **data** component.
- **Geometry**: The plot above is a scatterplot. This is referred to as the **geometry** component. Other possible geometries include barplot, histogram, smooth densities, qqplot, and boxplot. We will learn more about these in Section 8.14.
- **Aesthetic mapping**: The plot uses several visual cues to represent the information provided by the dataset. The two most important cues in this plot are the point positions on the x-axis and y-axis, which represent population size and the total number of murders, respectively. Each point represents a different observation, and we *map* data about these observations to visual cues like x- and y-scale. Color is another visual cue that we map to region. We refer to this as the **aesthetic mapping** component. How we define the mapping depends on what **geometry** we are using.

We also note that:

- The points are labeled with the state abbreviations.
- The range of the x-axis and y-axis appears to be defined by the range of the data. They are both on log-scales.
- There are labels, a title, a legend, and we use the style of The Economist magazine.

ISTUDY

The general approach in **ggplot2** is to construct the plot part by part by adding *layers* to a `ggplot` object, created by the `ggplot` function. Layers can define geometries, compute summary statistics, define what scales to use, or change styles. To add layers, we use the symbol `+`. In general, a line of code will look like this:

$$\text{DATA} \mid\!> \texttt{ggplot()} + \text{LAYER 1} + \text{LAYER 2} + \cdots + \text{LAYER N}$$

We will now illustrate the basics of ggplot2 by dissecting how we construct the plot above.

8.2 Initializing an object with data

We start by loading the relevant dataset which is in the **dslabs package**:

```
library(dslabs)
```

The first step in creating a **ggplot2** graph is to define a `ggplot` object. Typically most or all the layers will be mapping variables from the same dataset, so we associate this object with the relevant data frame

```
ggplot(data = murders)
```

or equivalently

```
murders |> ggplot()
```


Both these lines of code render a plot, in this case a blank slate since no geometry has been defined. The only style choice we see is a gray background, the default. We see a plot because an object was created and not assigned to a variable, it was automatically evaluated and printed. But we can assign our plot to an object in the usual way:

```
p <- ggplot(data = murders)
```

ISTUDY

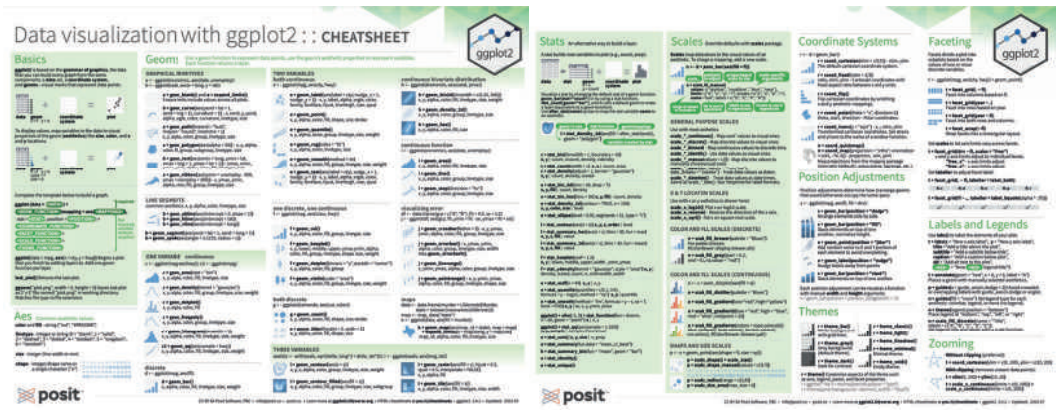
To render the plot associated with this object, we simply print the object `p`. The following two lines of code each produce the same plot we see above:

```
print(p)
p
```

To summarize, `p` is a `ggplot` object with the `murders` data frame as its data component.

8.3 Adding a geometry

A common first step is to let **ggplot2** know what geometry to use. We often add multiple geometries, but we at least need one. In our example, we want to make a scatterplot. Geometries are added using functions. Taking a quick look at the cheat sheet, we see that we should use the function `geom_point`.



(Image courtesy of Posit[4]. CC-BY-4.0 license[5].)

Note that geometry function names follow the pattern: `geom_X` where X is the name of the geometry. Some examples include `geom_histogram`, `geom_boxplot`, and `geom_col`. We will discuss these further in Section 8.14.

For `geom_point` to run properly we need to provide data and a mapping. We have already assigned the `murders` data table to the object `p`. Next we need to add the layer `geom_point` to define the geometry. To find out what mappings are expected by this function, we read the **Aesthetics** section of the `geom_point` help file:

[4] https://rstudio.github.io/cheatsheets/data-visualization.pdf
[5] https://github.com/rstudio/cheatsheets/blob/main/LICENSE

ISTUDY

Aesthetics
geom_point() understands the following aesthetics (required aesthetics are in bold):

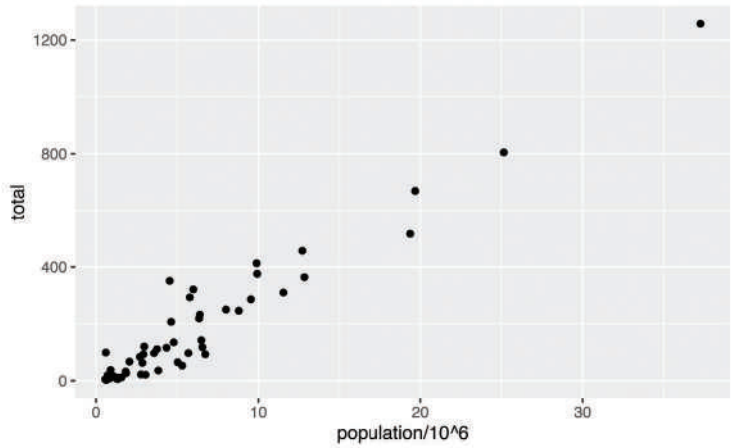
- **x**
- **y**
- alpha
- colour

We see that at least two arguments are required: `x` and `y`. Next we explain how to map values from the dataset to the plot.

8.4 Aesthetic mappings

Aesthetic mappings describe how properties of the data connect with features of the graph, such as distance along an axis, size, or color. The `aes` function connects data with what we see on the graph by defining aesthetic mappings and will be one of the functions you use most often when plotting. This example produces a scatterplot of total murders versus population in millions:

```
murders |> ggplot() + geom_point(aes(population/10^6, total))
```



We didn't use the `x` and `y` to define the arguments because the help file showed these are the first and second expected arguments.

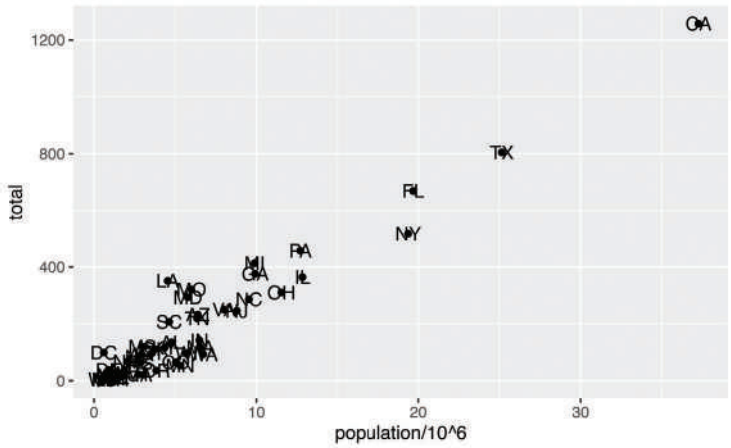
The scale and labels are defined by default when adding this layer. Like **dplyr** functions, `aes` also uses the variable names from the data component: we can use `population` and `total` without having to call them as `murders$population` and `murders$total`. The behavior of recognizing the variables from the data component is specific to `aes`. With **ggplot2** functions other than `aes`, if you try to access the values of `population` or `total`, you receive an error.

8.5 Other layers

To shape the plot into its final form, we continue to add layers. A second layer in the plot we wish to make involves adding a label to each point to identify the state. The `geom_label` and `geom_text` functions permit us to add text to the plot with and without a rectangle behind the text, respectively.

Because each point (each state in this case) has a label, we need an aesthetic mapping to make the connection between points and labels. By reading the help file, we learn that we supply the mapping between point and label through the `label` argument of `aes`. So the code looks like this:

```
murders |> ggplot() +
  geom_point(aes(population/10^6, total)) +
  geom_text(aes(population/10^6, total, label = abb))
```



As an example of the unique behavior of `aes` related to variable names, note that if the added layer was `geom_text(aes(population/10^6, total), label = abb)`, we would result in an error since `abb` is now outside the call to `aes` and it is not object in out workspace, it is a variable name in the data component of the `ggplot` object.

8.6 Global aesthetic mappings

In the previous lines of code, we define the mapping `aes(population/10^6, total)` twice, once in each layer. We can avoid this by using a *global* aesthetic mapping. We can do this when we define the blank slate `ggplot` object. Remember that the `mapping` argument of `ggplot` function permits us to define aesthetic mappings. If we define a mapping in `ggplot`, all the geometries that are added as layers will default to this mapping. So we can simply write the following code to produce the previous plot:

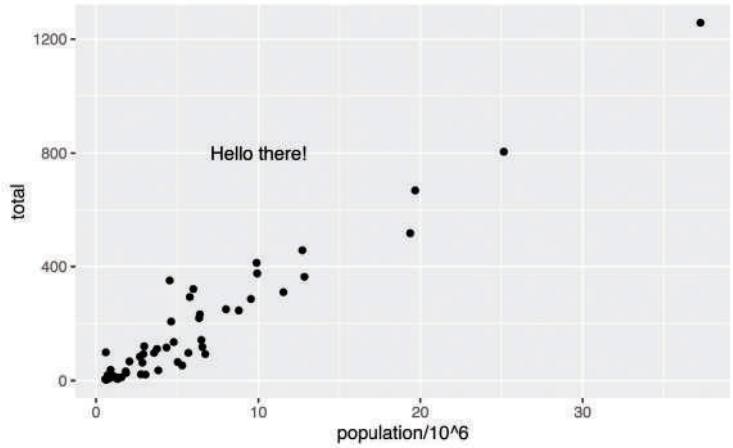
ISTUDY

```
murders |> ggplot(aes(population/10^6, total)) +
  geom_point() +
  geom_text(aes(label = abb))
```

Note that the mapping for `label` is only defined in `geom_text` because `geom_point` does not use this argument.

If necessary, we can override the global mapping by defining a new mapping within each layer. These *local* definitions override the *global*. Here is an example:

```
murders |> ggplot(aes(population/10^6, total)) +
  geom_point() +
  geom_text(aes(x = 10, y = 800, label = "Hello there!"))
```



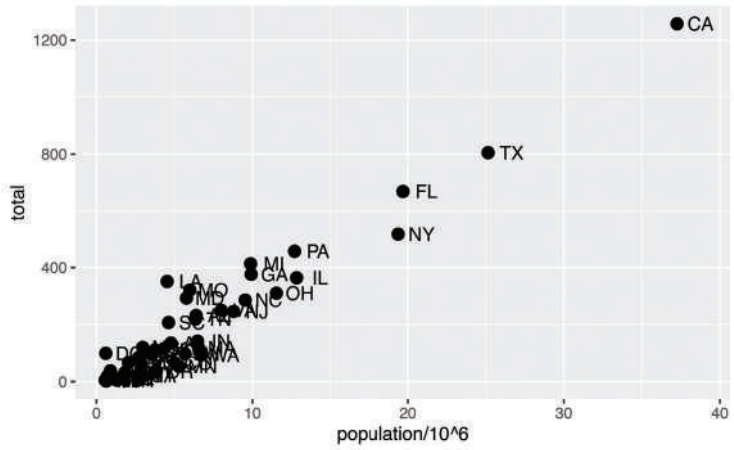
Clearly, the second call to `geom_text` does not use `population` and `total`.

8.7 Non-aesthetic arguments

Each geometry function has arguments other than `aes` and `data`. They tend to be specific to the function and are not mapped to variables in the data. For example, in the plot we wish to make, the points are larger than the default size. As another example, to avoid putting the text on top of the point, we can use the `nudge_x` argument in `geom_text`. The code, with the arguments, looks like this:

```
murders |> ggplot(aes(population/10^6, total)) +
  geom_point(size = 3) +
  geom_text(aes(label = abb), nudge_x = 1.5)
```

ISTUDY

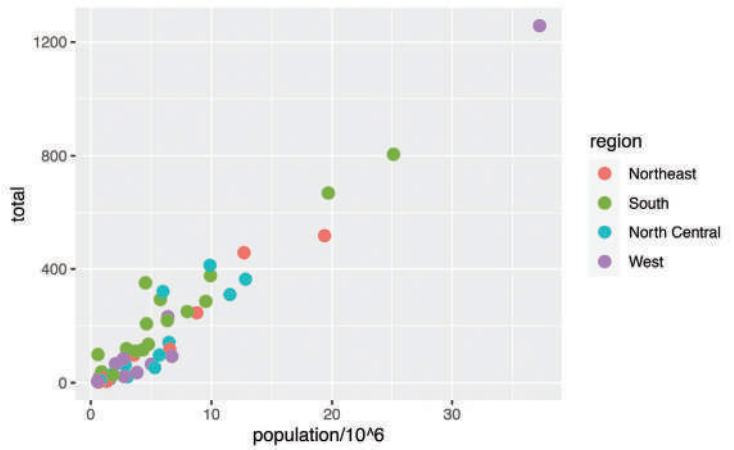


In Section 8.12 we learn a better way of assuring we can see the points and the labels.

8.8 Categories as colors

For the final plot, we want each region to have a different color. Because this information comes from the data, it is a aesthetic mapping. For our example, we can map color to categories using the `color` mapping in the `geom_point` function as follows:

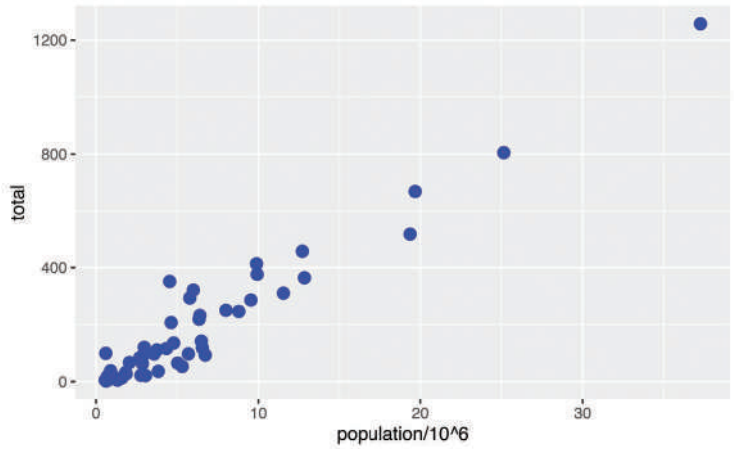
```
murders |> ggplot(aes(population/10^6, total)) +
  geom_point(aes(color = region), size = 3)
```



Note the `geom_point` automatically assigns a different color to each category and also adds a legend! Legends are usually desired, but to avoid adding a legend we can set the `geom_point` argument `show.legend = FALSE`.

Note that `color` is also a non-aesthetic argument in several **ggplot2** functions, including `geom_point`. This argument is not used to map colors to categories, but to change the color of all the points. For example, if we wanted all the points to be blue we would change the layer to `geom_point(col = "blue", size = 3)`.

```
murders |> ggplot(aes(population/10^6, total)) +
  geom_point(color = "blue", size = 3)
```



8.9 Updating ggplot objects

In **ggplot2** we build plots by parts. A useful feature of the package is that we can update existing `ggplot` objects by adding layers. For example, we can start by initializing an object with a dataset and a global aesthetic

```
p0 <- murders |> ggplot(aes(population/10^6, total))
```

and then start adding layers. For example, we start by adding the scatter plot

```
p1 <- p0 +  geom_point(aes(color = region), size = 3)
```

and labels:

```
p2 <- p1 + geom_text(aes(label = abb), nudge_x = 0.1)
```

In the next few sections, we will be building on objects created in previous sections using this approach. This facilitates improving plots as well as testing options. Note that we changed the `nudge_x` from 1.5 to 0.1 because in the next section we will apply a log transformation and a smaller value is more appropriate.

ISTUDY

8.10 Scales

One of the strengths of **ggplot2** is that the default behavior often is good enough to achieve our visualization goals. However, it also offers ways in which we can change these defaults. Many of these are changed through the `scales` functions.

Two examples, are the `scale_x_continuous` and `scale_y_continuous` functions which lets us make adjustments to the x-axis and y-axis, respectively. In the final plot we are trying to produce scales in log-scale and this can be achieved by assigning the argument `trans = "log10"` in these functions. However, because this operation is so common, **ggplot2** includes `scale_x_log10` and `scale_y_log10` functions. We can achieve the desired transformation by adding these layers:

```
p3 <- p2 + scale_x_log10() + scale_y_log10()
p3
```



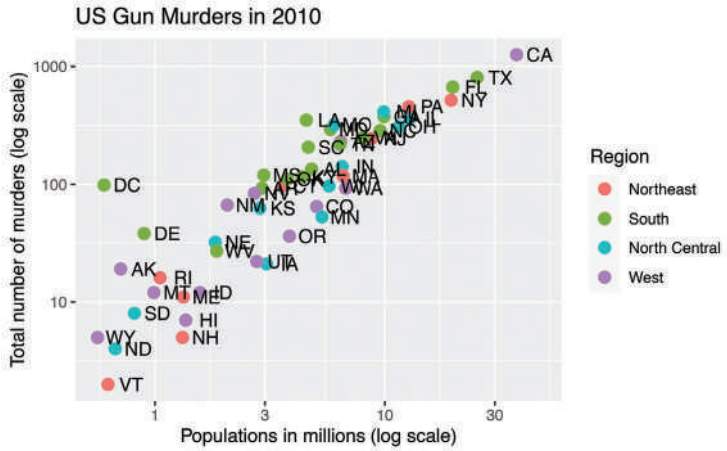
Be aware that **ggplot2** offers immense flexibility, particularly through the scales functions. We've introduced just one of the many available. In subsequent chapters of this book, we'll provide examples as they become pertinent to our visualizations. However, to familiarize yourself with these functions, we recommend consulting the ggplot2 cheat sheet or conducting internet searches as specific needs arise.

8.11 Annotations

We often want to add annotations to figures that are not derived directly from the aesthetic mapping. Examples of annotation functions are `labs`, `annotate`, and `geom_abline`. The `labs` function permits adding a title, subtitle, caption, and other labels. Note these can also be defined individually using the functions such as `xlab`, `ylab`, and `ggtitle`.

The `labs` function also allows another change needed for our desired plot: changing the legend title. Because the legend for the color mapping, this is achieved with the `color = "NEW_TITLE"` argument:

```
p4 <- p3 + labs(title = "US Gun Murders in 2010",
                x = "Populations in millions (log scale)",
                y = "Total number of murders (log scale)",
                color = "Region")
p4
```



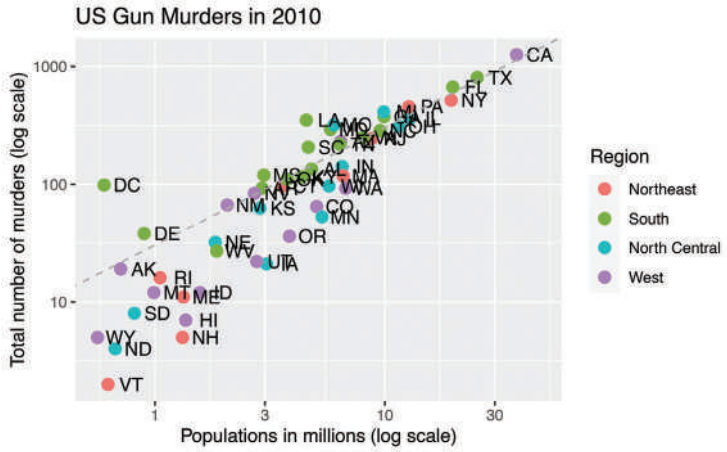
Our desired final plot includes a line that represents the average murder rate for the entire country. Once we determine the per million rate to be r, the desired line is defined by the formula: $y = rx$, with y and x our axes: total murders and population in millions, respectively. In the log-scale this line turns into: $\log(y) = \log(r) + \log(x)$, a line with slope 1 and intercept $\log(r)$. We can compute `r` using:

```
r <- murders |>
  summarize(rate = sum(total)/sum(population)*10^6) |>
  pull(rate)
```

To add a line we use the `geom_abline` function. The `ab` in the name reminds us we are supplying the intercept (`a`) and slope (`b`). The default line has slope 1 and intercept 0 so we only have to define the intercept. Note that the final plot has a dashed line type and is gray and these can be changed through the `lty` (line type) and `color` non aesthetic arguments. We add the layer like this:

```
p5 <- p4 +
  geom_abline(intercept = log10(r), lty = 2, color = "darkgrey")
p5
```

ISTUDY



Note that `geom_abline` does not use any mappings from the data object, once we have the slope.

We are almost there! All we have to do is add optional changes to the style.

8.12 Add-on packages

The power of **ggplot2** is augmented further due to the availability of add-on packages. The remaining changes needed to put the finishing touches on our plot require the **ggthemes** and **ggrepel** packages.

The style of a **ggplot2** graph can be changed using the `theme` functions. Several themes are included as part of the **ggplot2** package. In fact, for most of the plots in this book, we use a function in the **dslabs** package that automatically sets a default theme:

```
ds_theme_set()
```

Many other themes are added by the package **ggthemes**. Among those is the `theme_economist` theme that we use here. After installing the package, you can change the style by adding a layer like this:

```
library(ggthemes)
p6 <- p5 + theme_economist()
```

You can see how some of the other themes look by simply changing the function. For instance, you might try the `theme_fivethirtyeight()` theme instead.

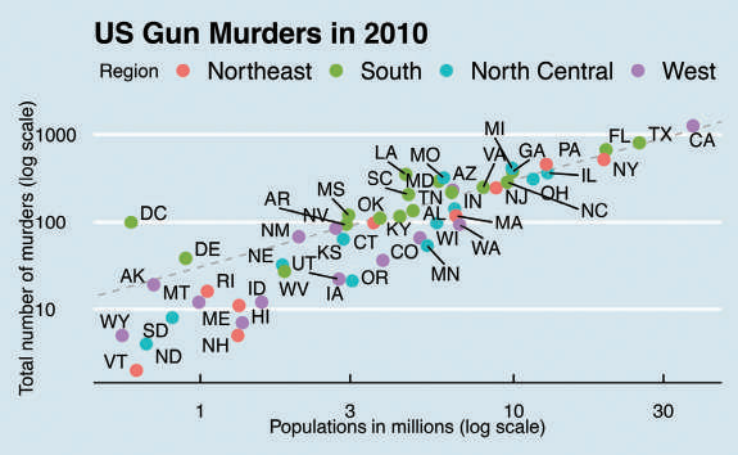
The final change is to better position of the labels to avoid crowding; currently, some of the labels fall on top of each other. The add-on package **ggrepel** includes a geometry that adds labels while ensuring that they don't fall on top of each other. We simply change `geom_text` to `geom_text_repel`.

ISTUDY

8.13 Putting it all together

Now that we are done testing, we can write one line of code that produces our desired plot from scratch.

```
library(ggthemes)
library(ggrepel)

r <- murders |>
  summarize(rate = sum(total) /  sum(population) * 10^6) |>
  pull(rate)

murders |>
  ggplot(aes(population/10^6, total)) +
  geom_abline(intercept = log10(r), lty = 2, color = "darkgrey") +
  geom_point(aes(col = region), size = 3) +
  geom_text_repel(aes(label = abb)) +
  scale_x_log10() +
  scale_y_log10() +
  labs(title = "US Gun Murders in 2010",
               x = "Populations in millions (log scale)",
               y = "Total number of murders (log scale)",
               color = "Region") +
  theme_economist()
```



8.14 Geometries

In our illustrative example we introduced the scatterplot geometry `geom_point`. However, **ggplot2** has many others and here we demonstrate how to generate plots related to distributions, specifically the plots shown Chapter 7.

8.14.1 Barplots

To generate a barplot we can use the `geom_bar` geometry. The default is to count the number of each category and draw a bar. Here is the plot for the regions of the US.

```
murders |> ggplot(aes(region)) + geom_bar()
```

However, we often already have a table with the numbers we want to present as a barplot. Here is an example of such a table:

```
tab <- murders |>
  count(region) |>
  mutate(proportion = n/sum(n))
```

In this case, we use `geom_col` instead of `geom_bar`:

```
tab |> ggplot(aes(region, proportion)) + geom_col()
```

8.14.2 Histograms

To generate histograms we use `geom_histogram`. By looking at the help file for this function, we learn that the only required argument is `x`, the variable for which we will construct a histogram. We dropped the `x` because we know it is the first argument. The code looks like this:

```
heights |> filter(sex == "Female") |>
  ggplot(aes(height)) +
  geom_histogram(binwidth = 1, fill = "blue", col = "black")
```

Note that we use the optional arguments `bandwidth = 1` to change the bin size to 1 inch. The default is to create 30 bins. We also use the optional arguments `fill = "blue"` and `col = "black"` to fill the bars with colors and use a different color to outline the bars.

8.14.3 Density plots

To create a smooth density, we use the `geom_density`. To make a smooth density plot with the data previously shown as a histogram we can use this code:

```
heights |>
  filter(sex == "Female") |>
  ggplot(aes(height)) +
  geom_density(fill = "blue")
```

ISTUDY

Note that we use the optional argument `fill` to change the color. To change the smoothness of the density, we use the `adjust` argument to multiply the default value by that `adjust`. For example, if we want the bandwidth to be twice as big we use:

```
heights |>
  filter(sex == "Female") |>
  ggplot(aes(height)) +
  geom_density(fill="blue", adjust = 2)
```

8.14.4 Boxplots

The geometry for boxplot is `geom_boxplot`. As discussed, boxplots are useful for comparing distributions. For example, below are the previously shown heights for women, but compared to men. For this geometry, we need arguments `x` as the categories, and `y` as the values.

```
heights |> ggplot(aes(sex, height)) +
  geom_boxplot()
```

8.14.5 Images

Images were not needed for the concepts described in this chapter, but we will use images in Section 10.9, so we introduce the two geometries used to plot images: **geom_tile** and **geom_raster**. They behave similarly; to see how they differ, please consult the help file. To create an image in **ggplot2** we need a data frame with the x and y coordinates as well as the values associated with each of these. Here is a data frame.

```
x <- expand.grid(x = 1:12, y = 1:10) |> mutate(z = 1:120)
```

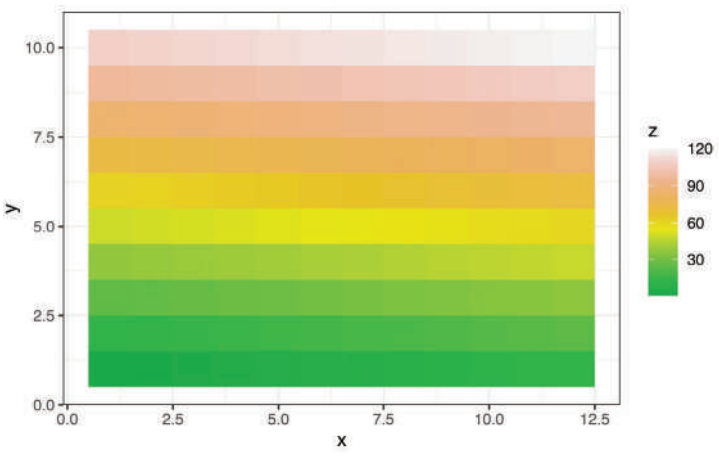
Note that this is the tidy version of a matrix, `matrix(1:120, 12, 10)`. To plot the image we use the following code:

```
x |> ggplot(aes(x, y, fill = z)) + geom_raster()
```

With these images you will often want to change the color scale. This can be done through the `scale_fill_gradientn` layer.

```
x |> ggplot(aes(x, y, fill = z)) +
  geom_raster() +
  scale_fill_gradientn(colors =  terrain.colors(10, 1))
```

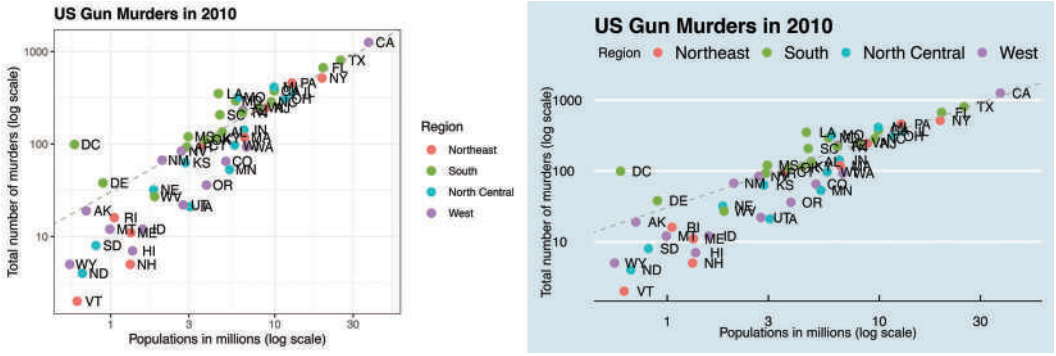
ISTUDY



8.15 Grids of plots

There are often reasons to graph plots next to each other. The **gridExtra** package permits us to do that. Here are the graphs `p5` and `p6` created in the previous sections:

```
library(gridExtra)
grid.arrange(p5, p6, ncol = 2)
```



8.16 Exercises

Start by loading the **dplyr** and **ggplot2** library as well as the `murders` and `heights` data.

```
library(dplyr)
library(ggplot2)
library(dslabs)
```

ISTUDY

1. With **ggplot2**, plots can be saved as objects. For example we can associate a dataset with a plot object like this

   ```
   p <- ggplot(data = murders)
   ```

   Because `data` is the first argument we don't need to spell it out

   ```
   p <- ggplot(murders)
   ```

   and we can also use the pipe:

   ```
   p <- murders |> ggplot()
   ```

   What is class of the object `p`?

2. Remember that to print an object you can use the command `print` or simply type the object. Print the object `p` defined in exercise one and describe what you see.

   (a) Nothing happens.
   (b) A blank slate plot.
   (c) A scatterplot.
   (d) A histogram.

3. Using the pipe `|>`, create an object `p` but this time associated with the `heights` dataset instead of the `murders` dataset.

4. What is the class of the object `p` you have just created?

5. Now we are going to add a layer and the corresponding aesthetic mappings. For the murders data we plotted total murders versus population sizes. Explore the `murders` data frame to remind yourself what are the names for these two variables and select the correct answer. **Hint**: Look at `?murders`.

   (a) `state` and `abb`.
   (b) `total_murders` and `population_size`.
   (c) `total` and `population`.
   (d) `murders` and `size`.

6. To create the scatterplot we add a layer with `geom_point`. The aesthetic mappings require us to define the x-axis and y-axis variables, respectively. So the code looks like this:

   ```
   murders |> ggplot(aes(x = , y = )) +
     geom_point()
   ```

   except we have to define the two variables `x` and `y`. Fill this out with the correct variable names.

ISTUDY

7. Note that if we don't use argument names, we can obtain the same plot by making sure we enter the variable names in the right order like this:

```
murders |> ggplot(aes(population, total)) +
  geom_point()
```

Remake the plot but now with total in the x-axis and population in the y-axis.

8. If instead of points we want to add text, we can use the `geom_text()` or `geom_label()` geometries. The following code

```
murders |> ggplot(aes(population, total)) + geom_label()
```

will give us the error message: `Error: geom_label requires the following missing aesthetics: label`

Why is this?

(a) We need to map a character to each point through the label argument in aes.
(b) We need to let `geom_label` know what character to use in the plot.
(c) The `geom_label` geometry does not require x-axis and y-axis values.
(d) `geom_label` is not a ggplot2 command.

9. Rewrite the code above to use abbreviation as the label through `aes`

10. Change the color of the labels to blue. How will we do this?

(a) Adding a column called `blue` to `murders`.
(b) Because each label needs a different color, we map the colors through `aes`.
(c) Use the `color` argument in `ggplot`.
(d) Because we want all labels to be blue, we do not need to map colors, just use the color argument in `geom_label`.

11. Rewrite the code above to make the labels blue.

12. Now suppose we want to use color to represent the different regions. In this case which of the following is most appropriate:

(a) Adding a column called `color` to `murders` with the color we want to use.
(b) Because each label needs a different color, we map the colors through the color argument of `aes` .
(c) Use the `color` argument in `ggplot`.
(d) Because we want all colors to be blue, we do not need to map colors, just use the color argument in `geom_label`.

13. Rewrite the code above to make the labels' colors be determined by the state's region.

14. Now we are going to change the x-axis to a log scale to account for the fact the distribution of population is skewed. Let's start by defining an object `p` holding the plot we have made up to now

```
p <- murders |>
  ggplot(aes(population, total, label = abb, color = region)) +
  geom_label()
```

ISTUDY

To change the y-axis to a log scale we learned about the `scale_x_log10()` function. Add this layer to the object `p` to change the scale and render the plot.

15. Repeat the previous exercise but now change both axes to be in the log scale.

16. Now edit the code above to add the title "Gun murder data" to the plot. Hint: use the `ggtitle` function.

17. Now we are going to use the `geom_histogram` function to make a histogram of the heights in the `height` data frame. When reading the documentation for this function we see that it requires just one mapping, the values to be used for the histogram. Make a histogram of all the plots.

    What is the variable containing the heights?

    (a) `sex`
    (b) `heights`
    (c) `height`
    (d) `heights$height`

18. Now create a ggplot object using the pipe to assign the heights data to a ggplot object. Assign `height` to the x values through the `aes` function.

19. Now we are ready to add a layer to actually make the histogram. Use the object created in the previous exercise and the `geom_histogram` function to make the histogram.

20. Note that when we run the code in the previous exercise we get the warning: `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

    Use the `binwidth` argument to change the histogram made in the previous exercise to use bins of size 1 inch.

21. Instead of a histogram, we are going to make a smooth density plot. In this case we will not make an object, but instead render the plot with one line of code. Change the geometry in the code previously used to make a smooth density instead of a histogram.

22. Now we are going to make a density plot for males and females separately. We can do this using the `group` argument. We assign groups via the aesthetic mapping as each point needs to a group before making the calculations needed to estimate a density.

23. We can also assign groups through the `color` argument. This has the added benefit that it uses colors to distinguish the groups. Change the code above to use color.

24. We can also assign groups through the `fill` argument. This has the added benefit that it uses colors to distinguish the groups, like this:

```
heights |>
  ggplot(aes(height, fill = sex)) +
  geom_density()
```

    However, here the second density is drawn over the other. We can make the curves more visible by using alpha blending to add transparency. Set the alpha parameter to 0.2 in the `geom_density` function to make this change.

ISTUDY

9

Data visualization principles

In this chapter we aim to provide some general principles we can use as a guide for effective data visualization. Much of this section is based on a talk by Karl Broman[1] titled "Creating Effective Figures and Tables"[2] and includes some of the figures which were made with code that Karl makes available on his GitHub repository[3], as well as class notes from Peter Aldhous' Introduction to Data Visualization course[4]. Following Karl's approach, we show some examples of plot styles we should avoid, explain how to improve them, and use these as motivation for a list of principles. We compare and contrast plots that follow these principles to those that don't.

The principles are mostly based on research related to how humans detect patterns and make visual comparisons. The preferred approaches are those that best fit the way our brains process visual information. When deciding on a visualization approach, it is also important to keep our goal in mind. We may be comparing a viewable number of quantities, describing distributions for categories or numeric values, comparing the data from two groups, or describing the relationship between two variables. As a final note, we want to emphasize that it is important to adapt and optimize graphs to the audience. For example, an exploratory plot made for ourselves will be different than a chart intended to communicate a finding to a general audience.

In this chapter we focus on the principles and do not show code (code can be viewed on GitHub[5]). In Chapter 10, we apply these principles in case study and do show code.

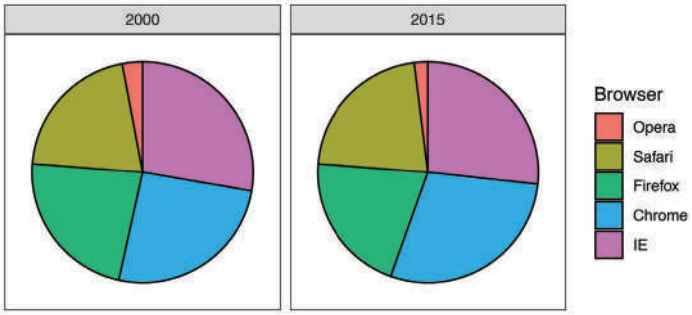
9.1 Encoding data using visual cues

We start by describing some principles for encoding data. There are several visual cues at our disposal including position, aligned lengths, angles, area, brightness, and color hue.

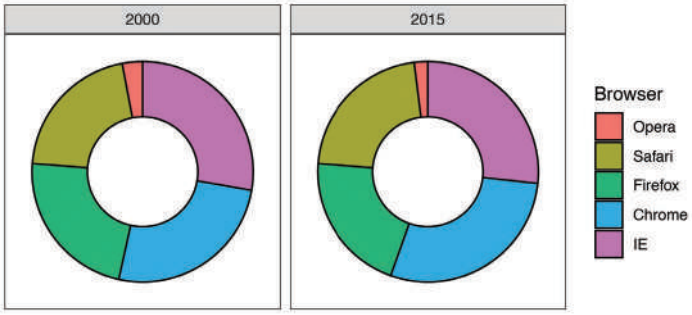
To illustrate how some of these visual cues compare, let's suppose we want to report the results from two hypothetical polls regarding browser preference taken in 2000 and then 2015. For each year, we are simply comparing five quantities – the five percentages. A widely used graphical representation of percentages, popularized by Microsoft Excel, is the pie chart:

[1] http://kbroman.org/
[2] https://www.biostat.wisc.edu/~kbroman/presentations/graphs2017.pdf
[3] https://github.com/kbroman/Talk_Graphs
[4] https://www.peteraldhous.com/ucb/2014/dataviz/index.html
[5] https://github.com/rafalab/dsbook-part-1/blob/main/dataviz/dataviz-principles.qmd

DOI: 10.1201/9781003220923-9



Here we are representing quantities with both areas and angles, since both the angle and area of each pie slice are proportional to the quantity the slice represents. This turns out to be a sub-optimal choice since, as demonstrated by perception studies, humans are not good at precisely quantifying angles and are even worse when area is the only available visual cue. The donut chart is an example of a plot that uses only area:



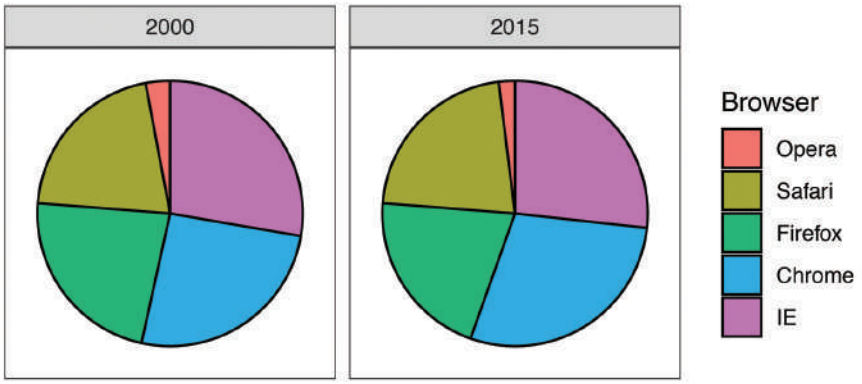
To see how hard it is to quantify angles and area, note that the rankings and all the percentages in the plots above changed from 2000 to 2015. Can you determine the actual percentages and rank the browsers' popularity? Can you see how the percentages changed from 2000 to 2015? It is not easy to tell from the plot.

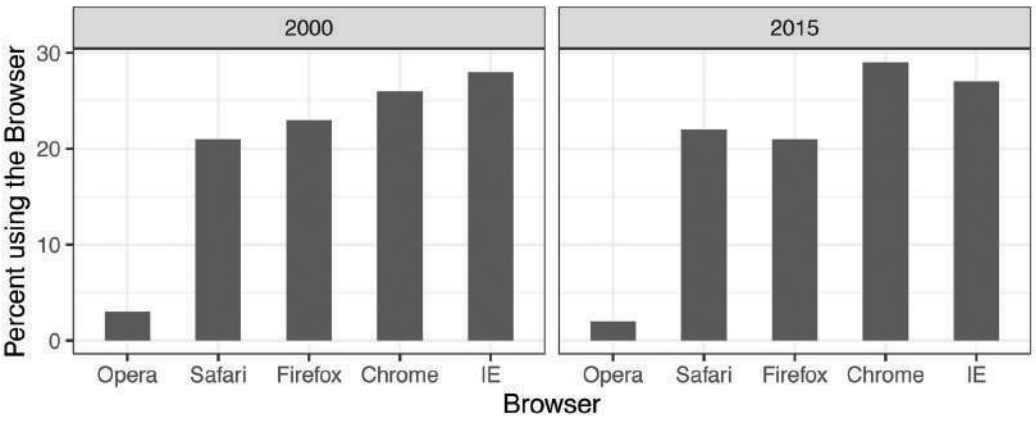
In this case, simply showing the numbers is not only clearer, but would also saves on printing costs if printing a paper copy:

ISTUDY

Browser	2000	2015
Opera	3	2
Safari	21	22
Firefox	23	21
Chrome	26	29
IE	28	27

The preferred way to plot these quantities is to use length and position as visual cues, since humans are much better at judging linear measures. The barplot uses this approach by using bars of length proportional to the quantities of interest. By adding horizontal lines at strategically chosen values, in this case at every multiple of 10, we ease the visual burden of quantifying through the position of the top of the bars. Compare and contrast the information we can extract from the two figures.

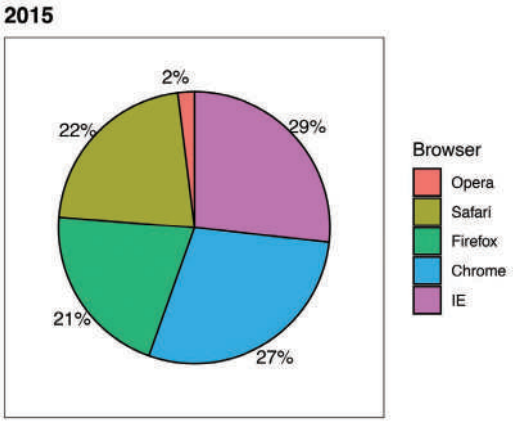




ISTUDY

Notice how much easier it is to see the differences in the barplot. In fact, we can now determine the actual percentages by following a horizontal line to the x-axis.

If for some reason you need to make a pie chart, label each pie slice with its respective percentage so viewers do not have to infer them from the angles or area:



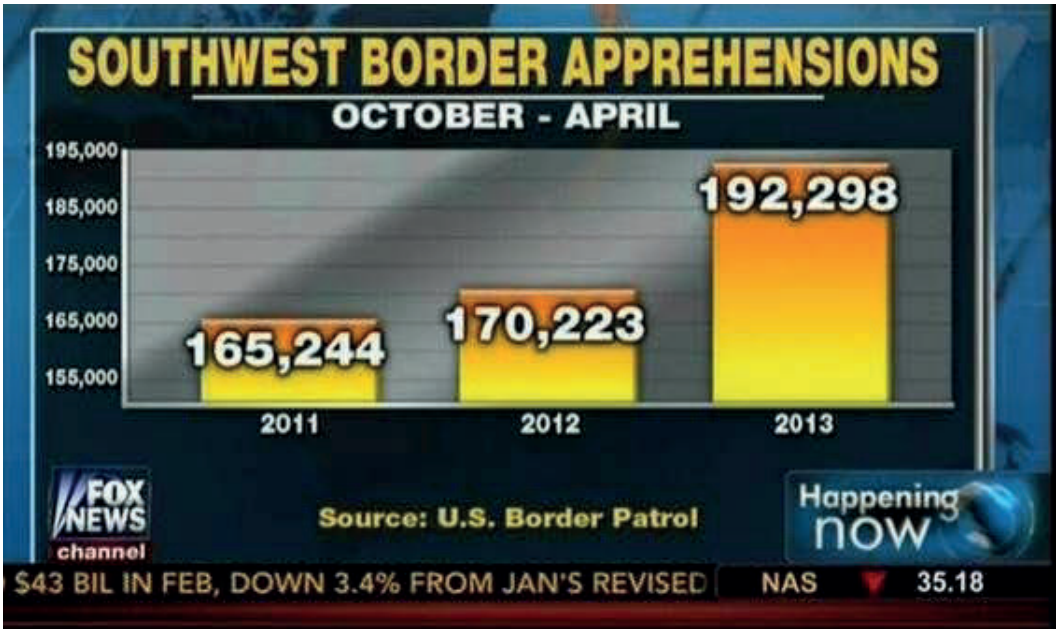
In general, when displaying quantities, position and length are preferred over angles and/or area. Brightness and color are even harder to quantify than angles. But, as we will see later, they are sometimes useful when more than two dimensions must be displayed at once.

9.2 Know when to include 0

When using length as a visual cue, it is misinformative not to start the bars at 0. This is because, by using length as a visual cue, say with a barplot, we are implying the length is proportional to the quantities being displayed. By avoiding 0, relatively small differences can be made to look much bigger than they actually are. This approach is often used by politicians or media organizations trying to exaggerate a difference. Below is an illustrative example[6]:

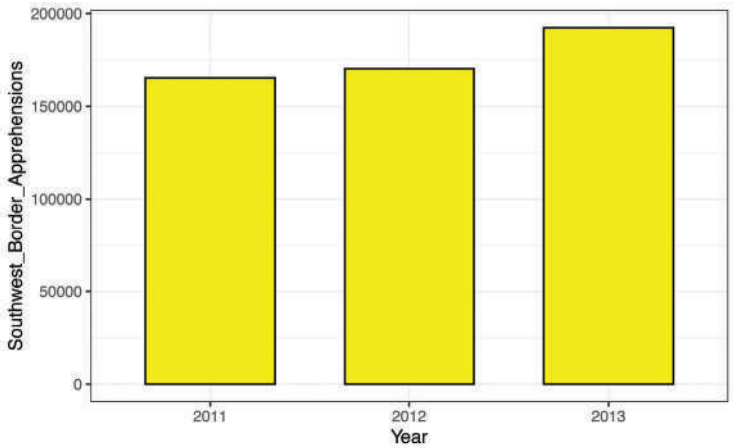
[6] https://www.peteraldhous.com/ucb/2014/dataviz/week2.html

ISTUDY



(Source: Fox News, via Media Matters[7].)

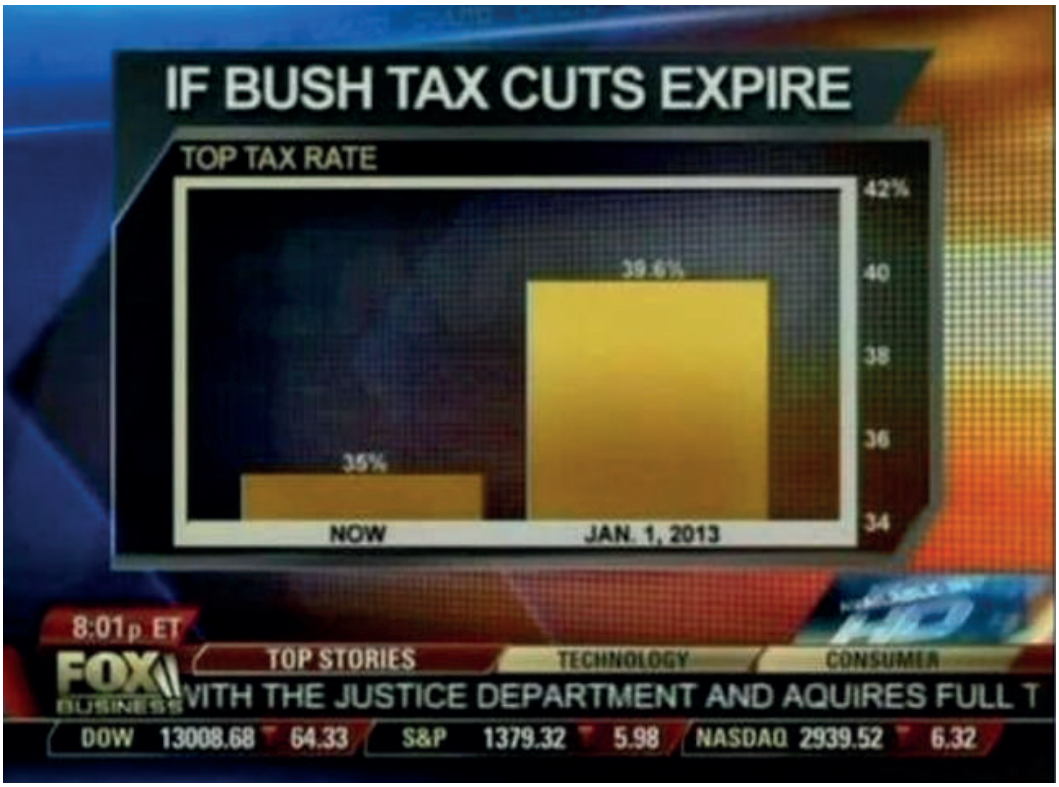
From the plot above, it appears that apprehensions have almost tripled when, in fact, they have only increased by about 16%. Starting the graph at 0 illustrates this clearly:



[7] http://mediamatters.org/blog/2013/04/05/fox-news-newest-dishonest-chart-immigration-enf/193507

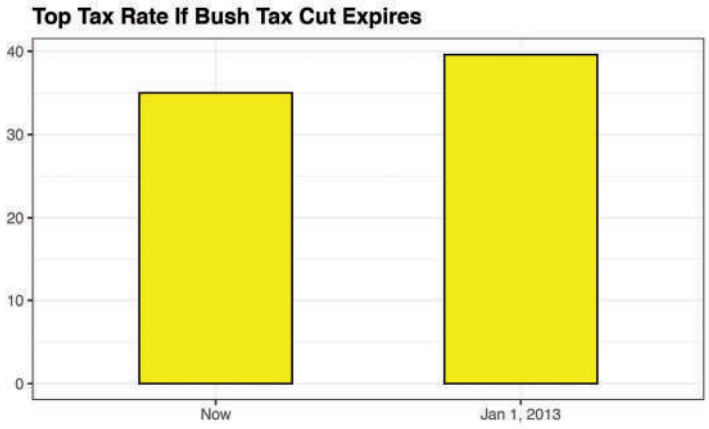
ISTUDY

Here is another example:



(Source: Fox News, via Flowing Data[8].)

This plot makes a 13% increase look like a five fold change. Here is the appropriate plot:



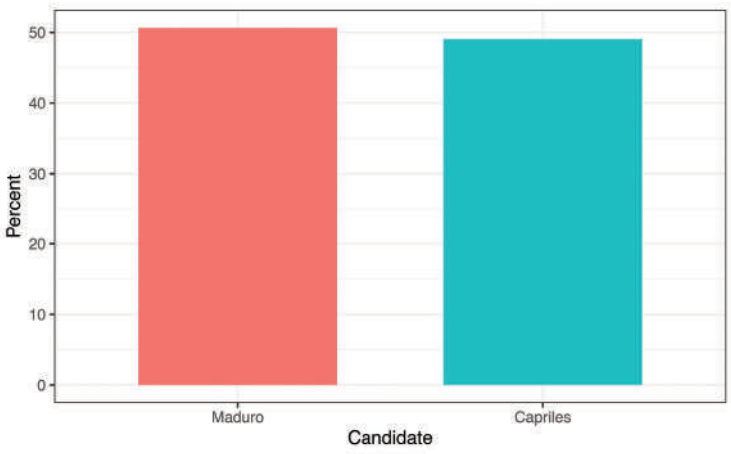
Finally, here is an extreme example that makes a very small difference of under 2% look like a 10–100 fold change:

[8] http://flowingdata.com/2012/08/06/fox-news-continues-charting-excellence/



(Source: Venezolana de Televisión via El Mundo[9].)

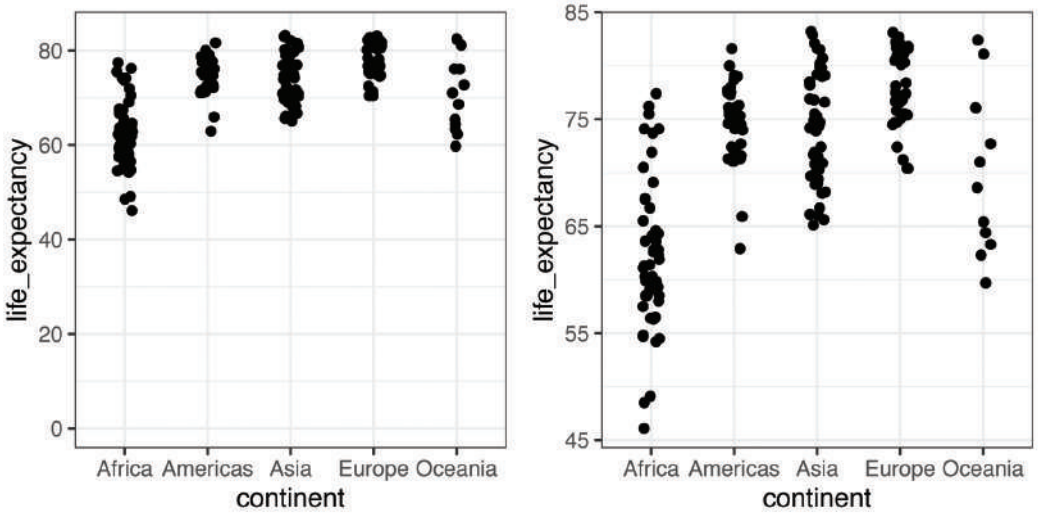
Here is the appropriate plot:



When using position rather than length, it is then not necessary to include 0. This is particularly the case when we want to compare differences between groups relative to the within-group variability. Here is an illustrative example showing country average life expectancy stratified across continents in 2012:

[9] https://www.elmundo.es/america/2013/04/15/venezuela/1366029653.html

ISTUDY



Note that in the plot on the left, which includes 0, the space between 0 and 43 adds no information and makes it harder to compare the between and within group variability.

9.3 Do not distort quantities

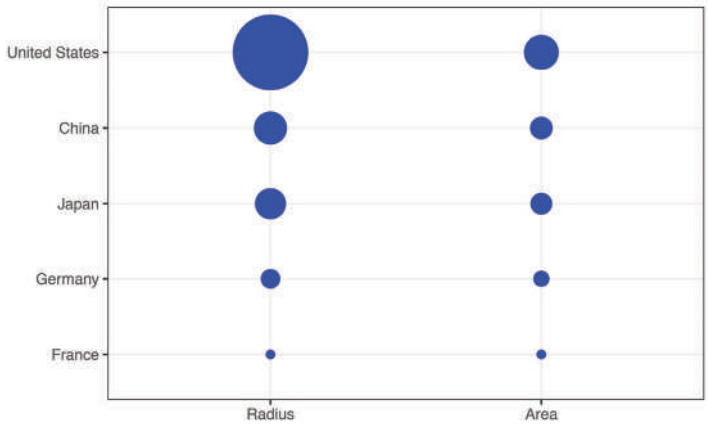
During President Barack Obama's 2011 State of the Union Address, the following chart was used to compare the US GDP to the GDP of four competing nations:



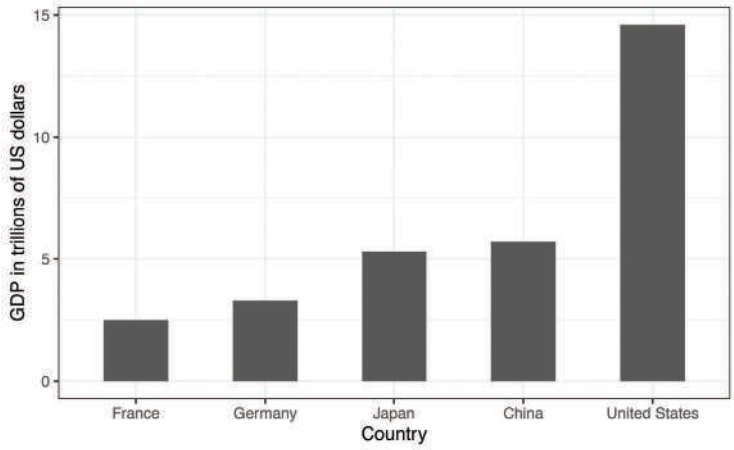
(Source: The 2011 State of the Union Address[10].)

[10] https://www.youtube.com/watch?v=kl2g40GoRxg

Judging by the area of the circles, the US appears to have an economy over five times larger than China's and over 30 times larger than France's. However, if we look at the actual numbers, we see that this is not the case. The actual ratios are 2.6 and 5.8 times bigger than China and France, respectively. The reason for this distortion is that the radius, rather than the area, was made to be proportional to the quantity, which implies that the proportion between the areas is squared: 2.6 turns into 6.5 and 5.8 turns into 34.1. Here is a comparison of the circles we get if we make the value proportional to the radius and to the area:



Not surprisingly, **ggplot2** defaults to using area rather than radius. Of course, in this case, we really should not be using area at all since we can use position and length:

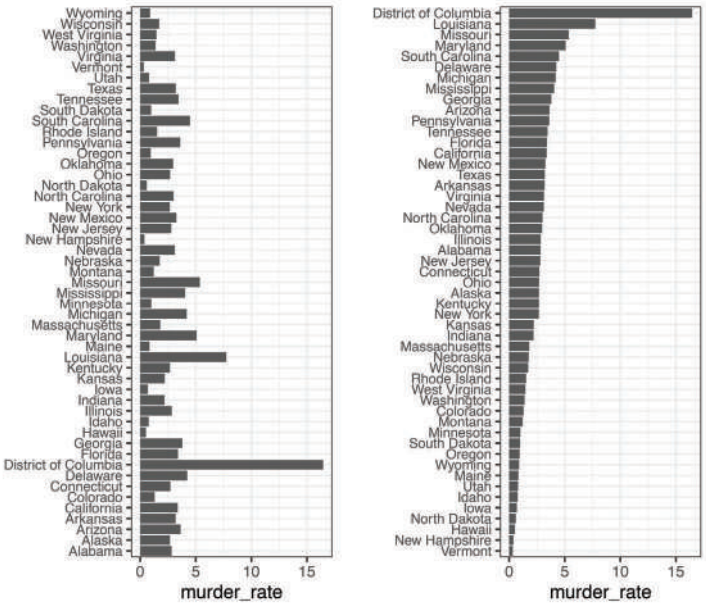


ISTUDY

9.4 Order categories by a meaningful value

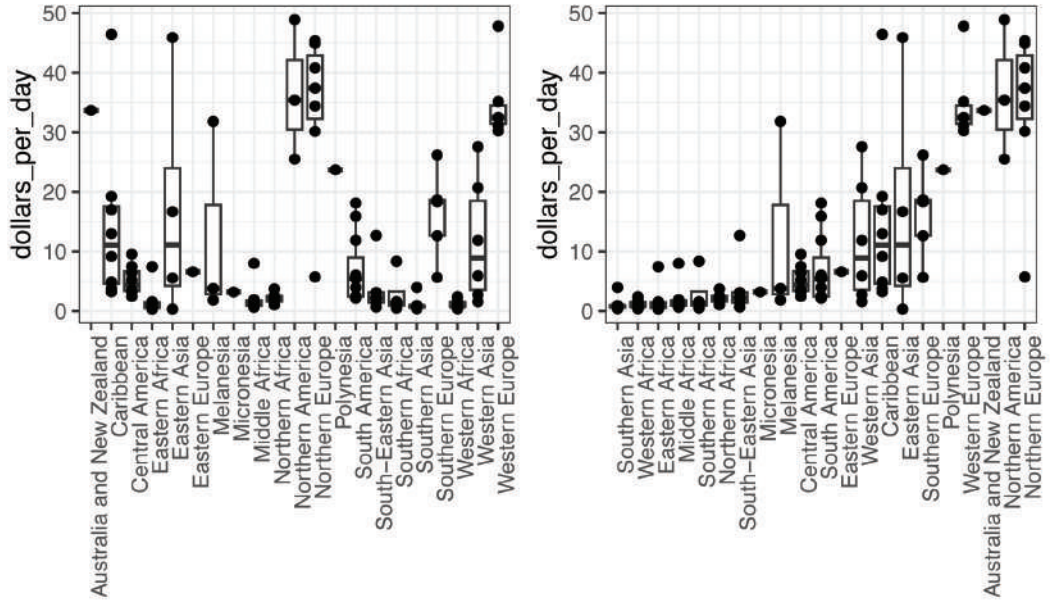
When one of the axes is used to show categories, as is done in barplots and boxplots, the default **ggplot2** behavior is to order the categories alphabetically when they are defined by character strings. If they are defined by factors, they are ordered by the factor levels. We rarely want to use alphabetical order. Instead, we should order by a meaningful quantity. In all the cases above, the barplots were ordered by the values being displayed. The exception was the graph showing barplots comparing browsers. In this case, we kept the order the same across the barplots to ease the comparison. Specifically, instead of ordering the browsers separately in the two years, we ordered both years by the average value of 2000 and 2015.

To appreciate how the right order can help convey a message, suppose we want to create a plot to compare the murder rate across states. We are particularly interested in the most dangerous and safest states. Note the difference when we order alphabetically (the default) versus when we order by the actual rate:



Here is an example showing boxplots of income distributions across regions. Here are the two versions plotted against each other:

ISTUDY



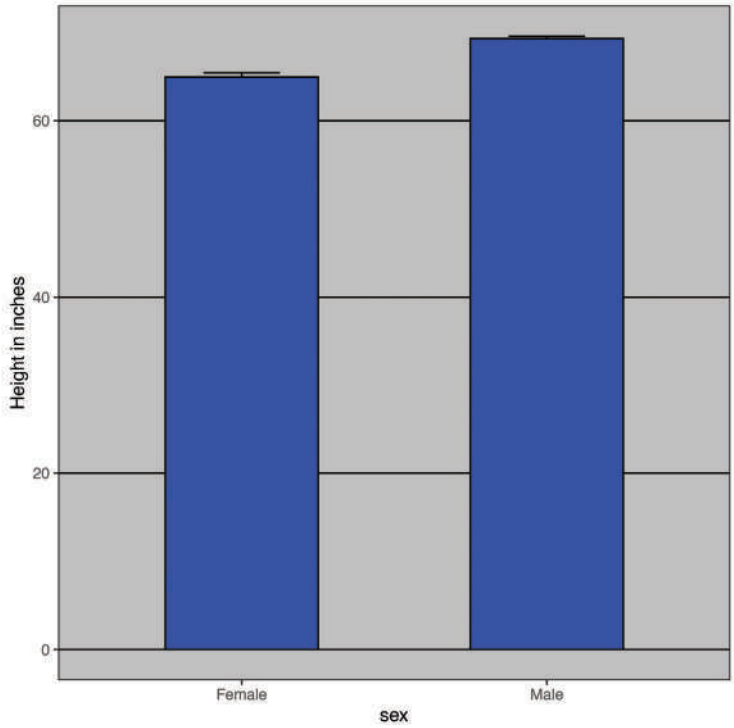
The first orders the regions alphabetically, while the second orders them by the group's median.

9.5 Show the data

We have focused on displaying single quantities across categories. We now shift our attention to displaying data, with a focus on comparing groups.

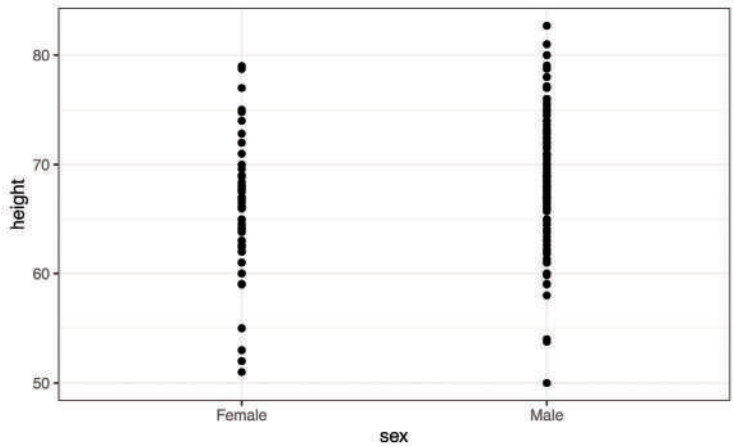
To motivate the principle, "show the data," we go back to our artificial example of describing heights to ET, an extraterrestrial. This time let's assume ET is interested in the difference in heights between males and females. A commonly seen plot used for comparisons between groups, popularized by software such as Microsoft Excel, is the dynamite plot, which shows the average and standard errors (standard errors are defined in a later chapter, but do not confuse them with the standard deviation of the data). The plot looks like this:

ISTUDY



The average of each group is represented by the top of each bar and the antennae extend out from the average to the average plus two standard errors. If all ET receives is this plot, he will have little information on what to expect if he meets a group of human males and females. The bars go to 0: does this mean there are tiny humans measuring less than one foot? Are all males taller than the tallest females? Is there a range of heights? ET can't answer these questions since we have provided almost no information on the height distribution.

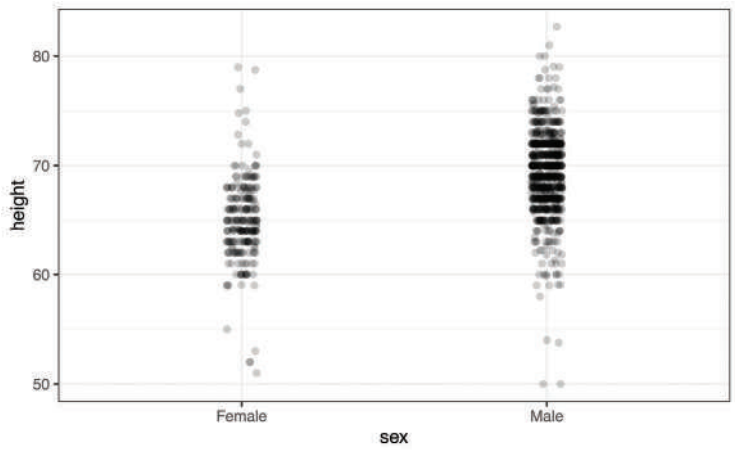
This brings us the "show the data" principle. This simple **ggplot2** code already generates a more informative plot than the barplot by simply showing all the data points:



ISTUDY

For example, this plot gives us an idea of the range of the data. However, this plot has limitations as well, since we can't really see all the 238 and 812 points plotted for females and males, respectively, and many points are plotted on top of each other. As we have previously described, visualizing the distribution is much more informative. But before doing this, we point out two ways we can improve a plot showing all the points.

The first is to add *jitter*, which adds a small random shift to each point. In this case, adding horizontal jitter does not alter the interpretation, since the point heights do not change, but we minimize the number of points that fall on top of each other and, therefore, get a better visual sense of how the data is distributed. A second improvement comes from using *alpha blending*: making the points somewhat transparent. The more points fall on top of each other, the darker the plot, which also helps us get a sense of how the points are distributed. Here is the same plot with jitter and alpha blending:



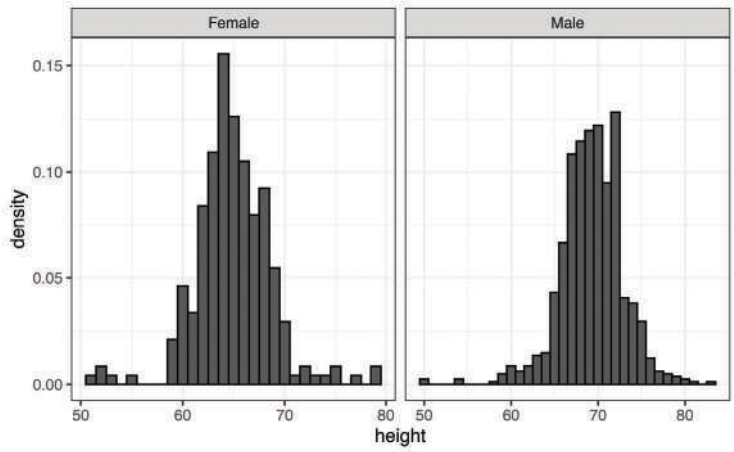
Now we start getting a sense that, on average, males are taller than females. We also note dark horizontal bands of points, demonstrating that many report values that are rounded to the nearest integer.

9.6 Ease comparisons

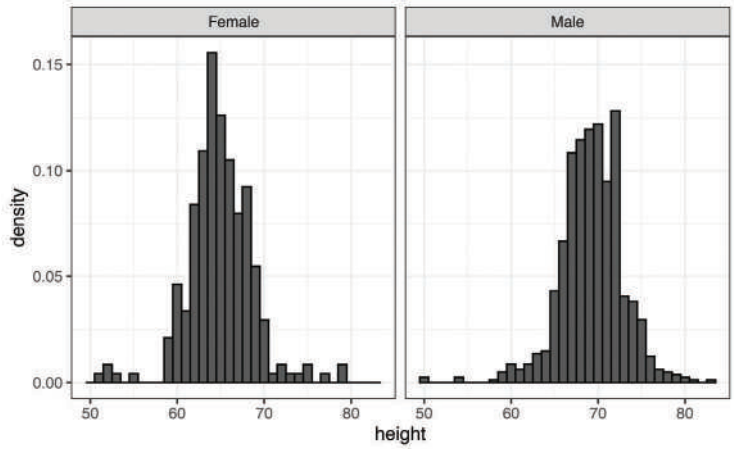
9.6.1 Use common axes

Since there are so many points, it is more effective to show distributions rather than individual points. We therefore show histograms for each group:

ISTUDY



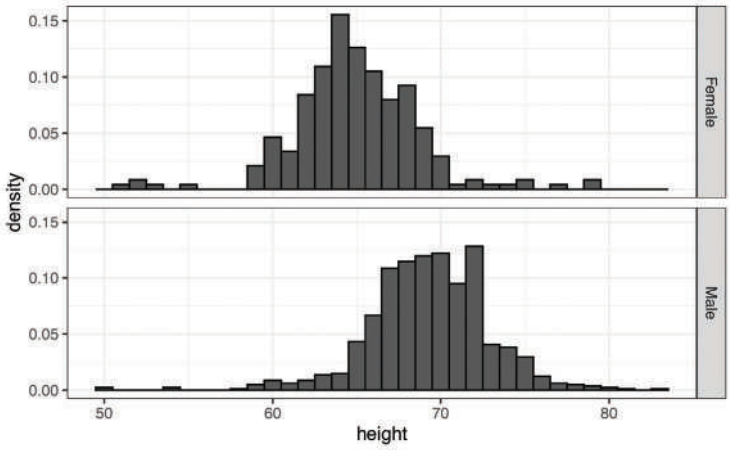
However, from this plot it is not immediately obvious that males are, on average, taller than females. We have to look carefully to notice that the x-axis has a higher range of values in the male histogram. An important principle here is to **keep the axes the same** when comparing data across two plots. Below we see how the comparison becomes a little easier:



9.6.2 Aligning plots for comparisons

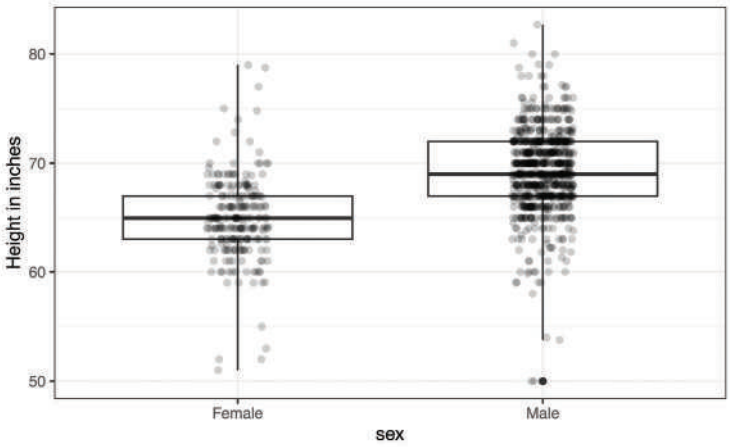
In these histograms, the visual cue related to decreases or increases in height are shifts to the left or right, respectively: horizontal changes. Aligning the plots vertically helps us see this change when the axes are fixed:

ISTUDY



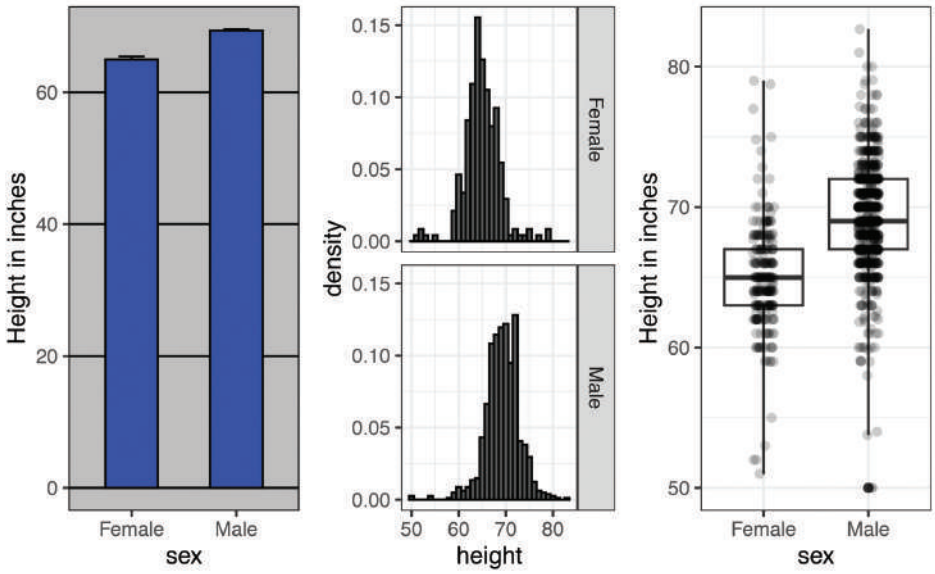
This plot makes it much easier to notice that men are, on average, taller.

If we want the more compact summary provided by boxplots, we then align them horizontally since, by default, boxplots move up and down with changes in height. Following our *show the data* principle, we then overlay all the data points:



Now contrast and compare these three plots, based on exactly the same data:

ISTUDY

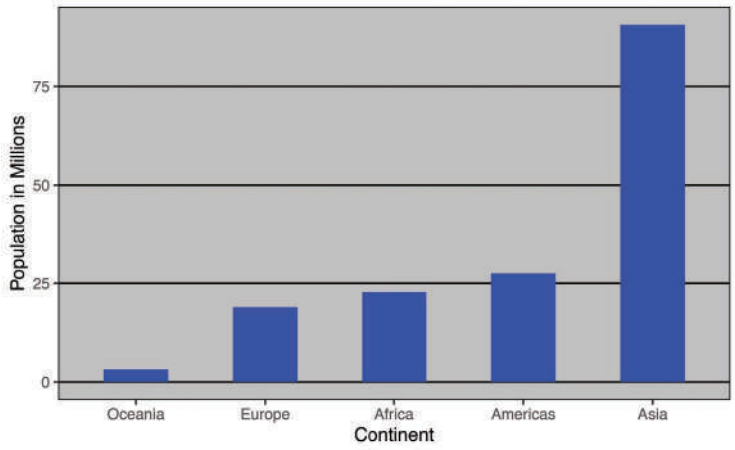


Notice how much more we learn from the two plots on the right. Barplots are useful for showing one number, but not very useful when we want to describe distributions.

9.7 Transformations

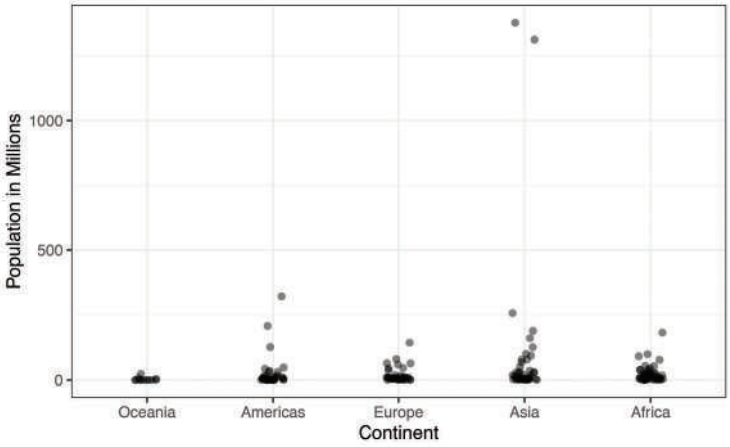
We have motivated the use of the log transformation in cases where the changes are multiplicative. Population size was an example in which we found a log transformation to yield a more informative transformation.

The combination of an incorrectly chosen barplot and a failure to use a log transformation when one is merited can be particularly distorting. As an example, consider this barplot showing the average population sizes for each continent in 2015:

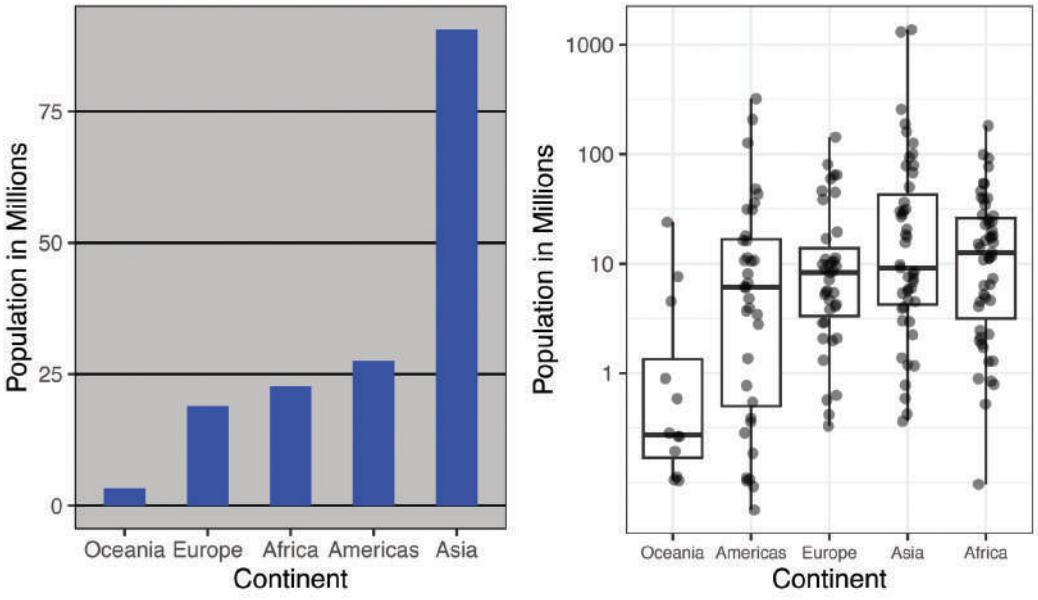


ISTUDY

From this plot, one would conclude that countries in Asia are much more populous than in other continents. Following the *show the data* principle, we quickly notice that this is due to two very large countries, which we assume are India and China:



Using a log transformation here provides a much more informative plot. We compare the original barplot to a boxplot using the log scale transformation for the y-axis:



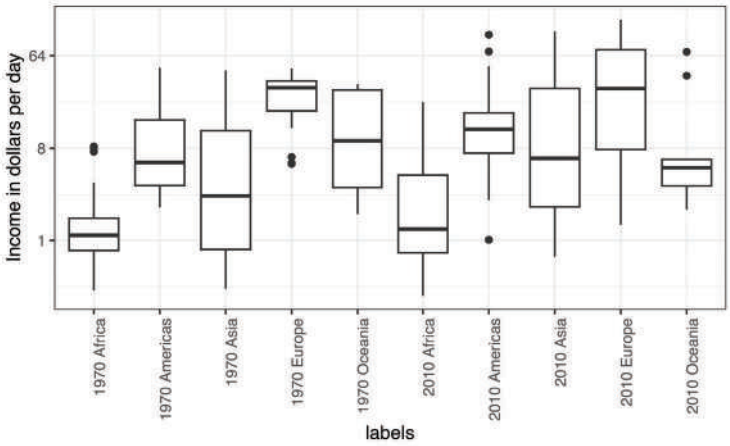
With the new plot, we realize that countries in Africa actually have a larger median population size than those in Asia.

Other transformations you should consider are the logistic transformation (`logit`), useful to better see fold changes in odds, and the square root transformation (`sqrt`), useful for count data.

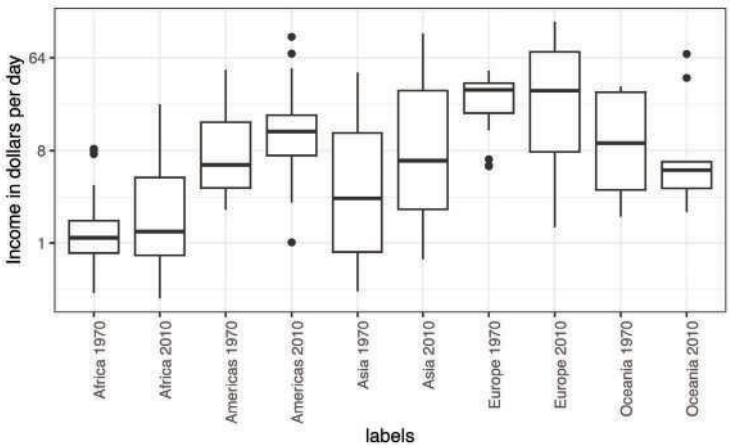
ISTUDY

9.8 Visual cues to be compared should be adjacent

For each continent, let's compare income in 1970 versus 2010. When comparing income data across regions between 1970 and 2010, we made a figure similar to the one below, but this time we investigate continents rather than regions.

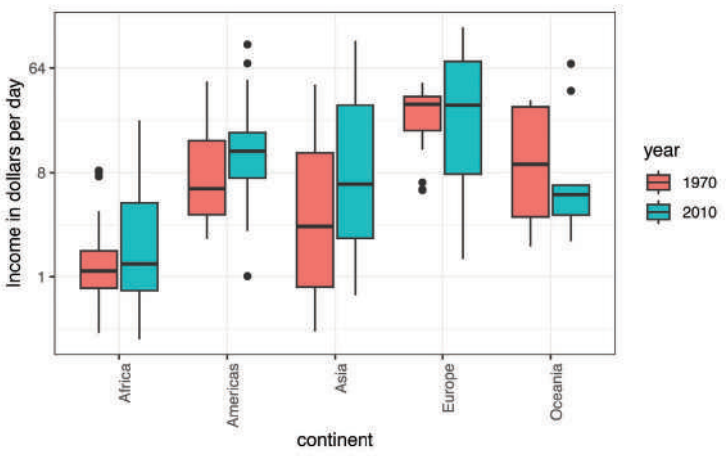


The default in **ggplot2** is to order labels alphabetically so the labels with 1970 come before the labels with 2010, making the comparisons challenging because a continent's distribution in 1970 is visually far from its distribution in 2010. It is much easier to make the comparison between 1970 and 2010 for each continent when the boxplots for that continent are next to each other:



The comparison becomes even easier to make if we use color to denote the two things we want to compare:

ISTUDY



9.9 Think of the color blind

About 10% of the population is color blind. Unfortunately, the default colors used in **ggplot2** are not optimal for this group. However, **ggplot2** does make it easy to change the color palette used in the plots. An example of how we can use a color blind friendly palette is described in the R cookbook[11]:

```
color_blind_friendly_cols <-
  c("#999999", "#E69F00", "#56B4E9", "#009E73",
    "#F0E442", "#0072B2", "#D55E00", "#CC79A7")
```

Here are the colors



There are several resources that can help you select colors, for example tutorials on R-bloggers[12].

9.10 Plots for two variables

In general, you should use scatterplots to visualize the relationship between two variables. In every single instance in which we have examined the relationship between two variables, including total murders versus population size and life expectancy versus fertility rates, we have used scatterplots. This is the plot we generally recommend. However, there are some exceptions and we describe two alternative plots here: the *slope chart* and the *Bland-Altman plot*.

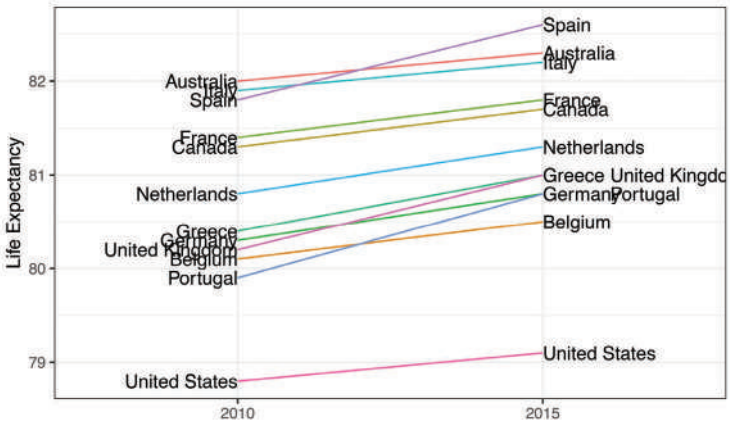
[11] http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette
[12] https://www.r-bloggers.com/2013/10/creating-colorblind-friendly-figures/

ISTUDY

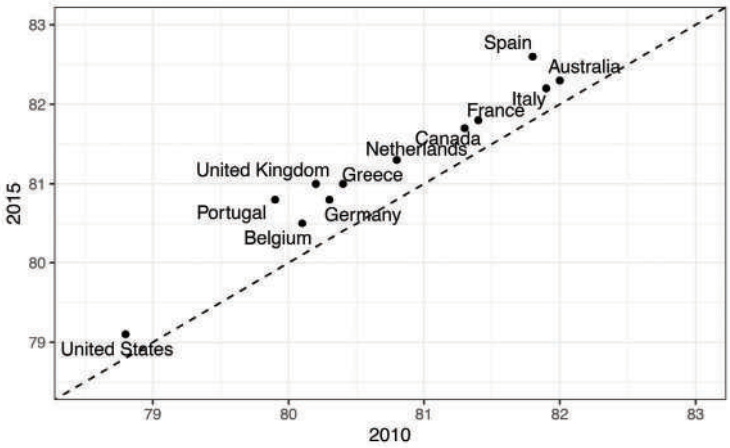
9.10.1 Slope charts

One exception where another type of plot may be more informative is when you are comparing variables of the same type, but at different time points and for a relatively small number of comparisons. For example, comparing life expectancy between 2010 and 2015. In this case, we might recommend a *slope chart*.

There is no geometry for slope charts in **ggplot2**, but we can construct one using `geom_line`. We need to do some tinkering to add labels. Below is an example comparing 2010–2015 for large western countries:



An advantage of the slope chart is that it permits us to quickly get an idea of changes based on the slope of the lines. Although we are using angle as the visual cue, we also have position to determine the exact values. Comparing the improvements is a bit harder with a scatterplot:

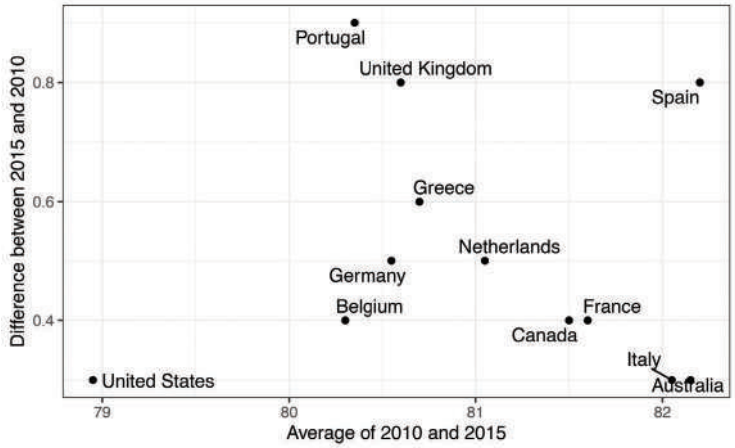


In the scatterplot, we have followed the principle *use common axes* since we are comparing these before and after. However, if we have many points, slope charts stop being useful as it becomes hard to see all the lines.

ISTUDY

9.10.2 Bland-Altman plot

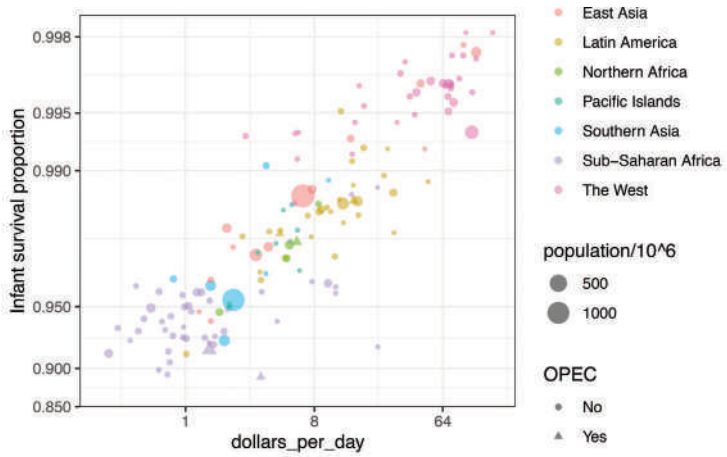
Since we are primarily interested in the difference, it makes sense to dedicate one of our axes to it. The Bland-Altman plot, also known as the Tukey mean-difference plot and the MA-plot, shows the difference versus the average:



Here, by simply looking at the y-axis, we quickly see which countries have shown the most improvement. We also get an idea of the overall value from the x-axis.

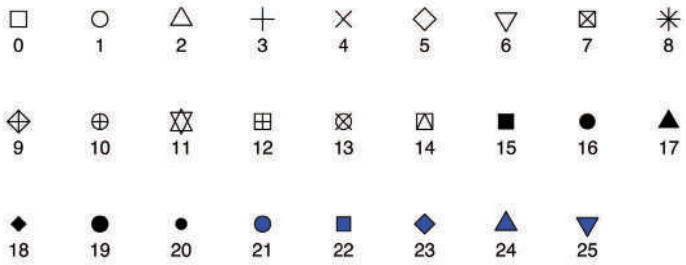
9.11 Encoding a third variable

An earlier scatterplot showed the relationship between infant survival and average income. Below is a version of this plot that encodes three additional variables: OPEC membership, region, and population.



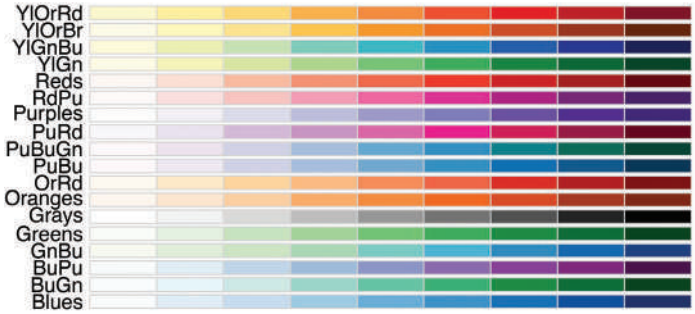
ISTUDY

We encode categorical variables with color and shape. These shapes can be controlled with `shape` argument. Below are the shapes available for use in R. For the last five, the color goes inside.

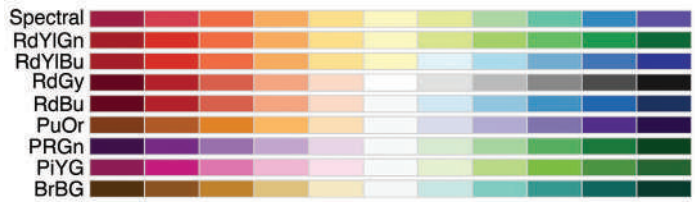


For continuous variables, we can use color, intensity, or size. We now show an example of how we do this with a case study.

When selecting colors to quantify a numeric variable, we choose between two options: sequential and diverging. Sequential colors are suited for data that goes from high to low. High values are clearly distinguished from low values. Here are some examples offered by the package `RColorBrewer`:



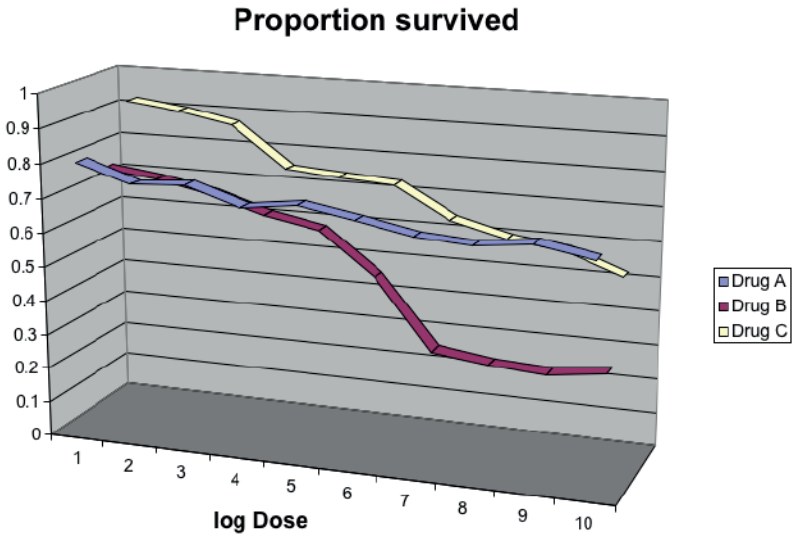
Diverging colors are used to represent values that diverge from a center. We put equal emphasis on both ends of the data range: higher than the center and lower than the center. An example of when we would use a divergent pattern would be if we were to show height in standard deviations away from the average. Here are some examples of divergent patterns:



ISTUDY

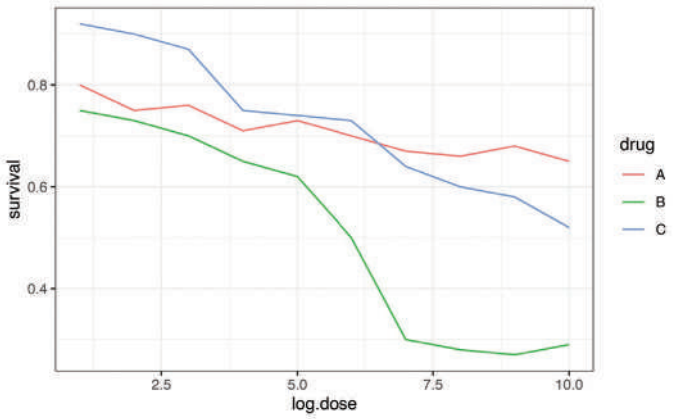
9.12 Avoid pseudo-three-dimensional plots

The figure below, taken from the scientific literature[13], shows three variables: dose, drug type and survival. Although your screen/book page is flat and two-dimensional, the plot tries to imitate three dimensions and assigned a dimension to each variable.



(Image courtesy of Karl Broman)

Humans are not good at seeing in three dimensions (which explains why it is hard to parallel park) and our limitation is even worse with regard to pseudo-three-dimensions. To see this, try to determine the values of the survival variable in the plot above. Can you tell when the purple ribbon intersects the red one? This is an example in which we can easily use color to represent the categorical variable instead of using a pseudo-3D:

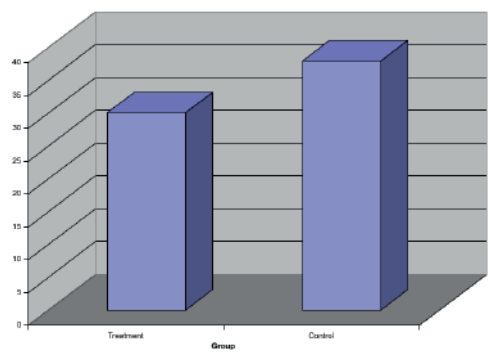


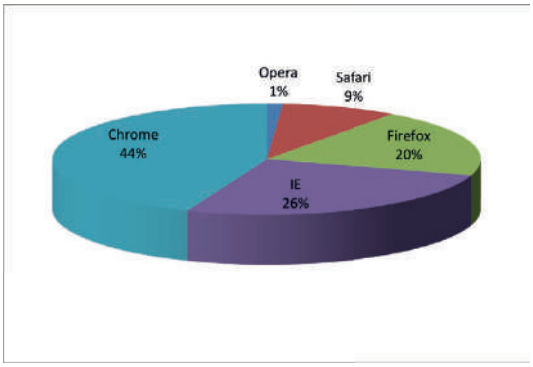
Notice how much easier it is to determine the survival values.

[13] https://projecteuclid.org/download/pdf_1/euclid.ss/1177010488

ISTUDY

Pseudo-3D is sometimes used completely gratuitously: plots are made to look 3D even when the 3rd dimension does not represent a quantity. This only adds confusion and makes it harder to relay your message. Here are two examples:





(Images courtesy of Karl Broman)

9.13 Avoid too many significant digits

By default, statistical software like R returns many significant digits. The default behavior in R is to show 7 significant digits. That many digits often adds no information and the added visual clutter can make it hard for the viewer to understand the message. As an example, here are the per 10,000 disease rates, computed from totals and population in R, for California across the five decades:

state	year	Measles	Pertussis	Polio
California	1940	37.8826320	18.3397861	0.8266512
California	1950	13.9124205	4.7467350	1.9742639
California	1960	14.1386471	NA	0.2640419
California	1970	0.9767889	NA	NA
California	1980	0.3743467	0.0515466	NA

We are reporting precision up to 0.00001 cases per 10,000, a very small value in the context of the changes that are occurring across the dates. In this case, one significant figures is enough and clearly makes the point that rates are decreasing:

state	year	Measles	Pertussis	Polio
California	1940	37.9	18.3	0.8
California	1950	13.9	4.7	2.0
California	1960	14.1	NA	0.3
California	1970	1.0	NA	NA
California	1980	0.4	0.1	NA

Useful ways to change the number of significant digits or to round numbers are `signif` and `round`. You can define the number of significant digits globally by setting options like this: `options(digits = 3)`.

ISTUDY

Another principle related to displaying tables is to place values being compared on columns rather than rows. Note that our table above is easier to read than this one:

State	Disease	1940	1950	1960	1970	1980
California	Measles	37.9	13.9	14.1	1	0.4
California	Pertussis	18.3	4.7	NA	NA	0.1
California	Polio	0.8	2.0	0.3	NA	NA

9.14 Know your audience

Graphs can be used 1) for our own exploratory data analysis, 2) to convey a message to experts, or 3) to help convey a message to a general audience. Make sure that the intended audience understands each element of the plot.

As a simple example, consider that for your own exploration it may be more useful to log-transform data and then plot it. However, for a general audience that is unfamiliar with converting logged values back to the original measurements, using a log-scale for the axis instead of log-transformed values will be much easier to digest.

9.15 Exercises

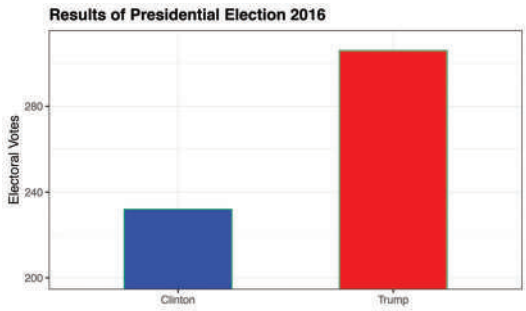
For these exercises, we will be using the vaccines data in the **dslabs** package:

```
library(dslabs)
```

1. Pie charts are appropriate:

   (a) When we want to display percentages.
   (b) When **ggplot2** is not available.
   (c) When I am in a bakery.
   (d) Never. Barplots and tables are always better.

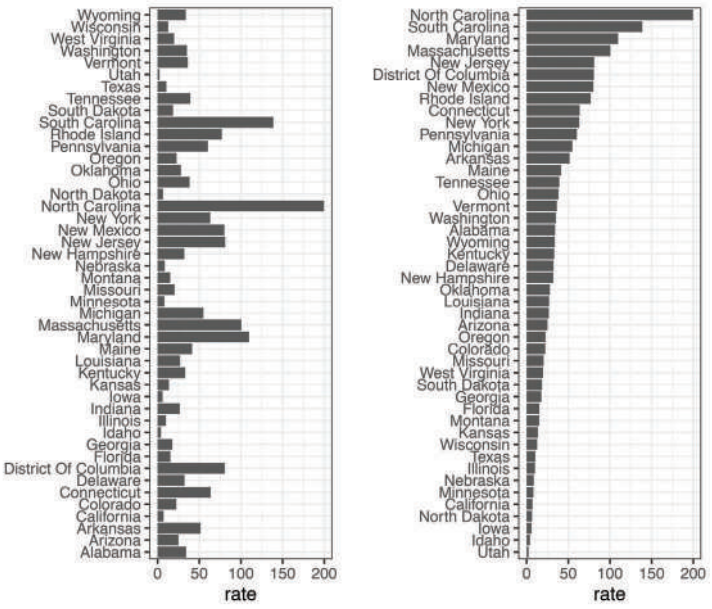
2. What is the problem with the plot below:



ISTUDY

(a) The values are wrong. The final vote was 306 to 232.
(b) The axis does not start at 0. Judging by the length, it appears Trump received 3 times as many votes when, in fact, it was about 30% more.
(c) The colors should be the same.
(d) Percentages should be shown as a pie chart.

3. Take a look at the following two plots. They show the same information: 1928 rates of measles across the 50 states.



Which plot is easier to read if you are interested in determining which are the best and worst states in terms of rates, and why?

(a) They provide the same information, so they are both equally as good.
(b) The plot on the right is better because it orders the states alphabetically.
(c) The plot on the right is better because alphabetical order has nothing to do with the disease and by ordering according to actual rate, we quickly see the states with most and least rates.
(d) Both plots should be a pie chart.

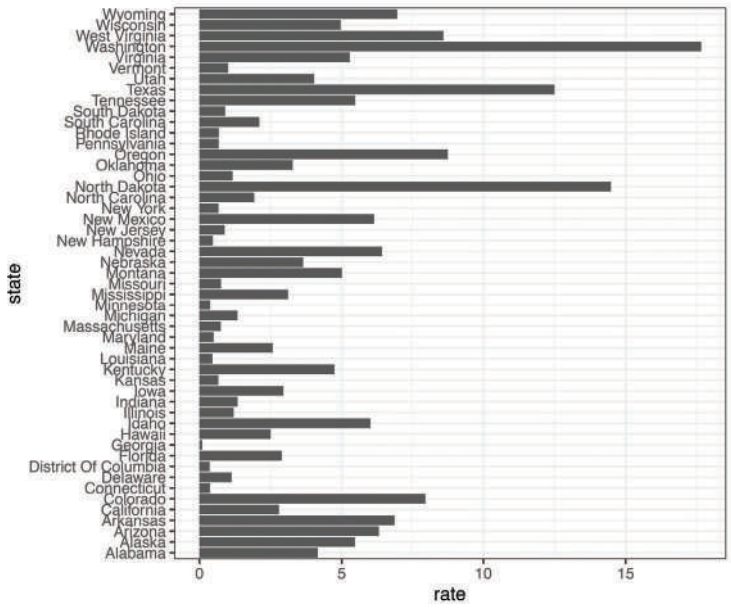
4. To make the plot on the left, we have to reorder the levels of the states' variables.

```
dat <- us_contagious_diseases |>
  filter(year == 1967 & disease=="Measles" & !is.na(population)) |>
  mutate(rate = count / population * 10000 * 52 / weeks_reporting)
```

Note what happens when we make a barplot:

```
dat |> ggplot(aes(state, rate)) +
  geom_col() +
  coord_flip()
```

ISTUDY



Define these objects:

```
state <- dat$state
rate <- dat$count/dat$population*10000*52/dat$weeks_reporting
```

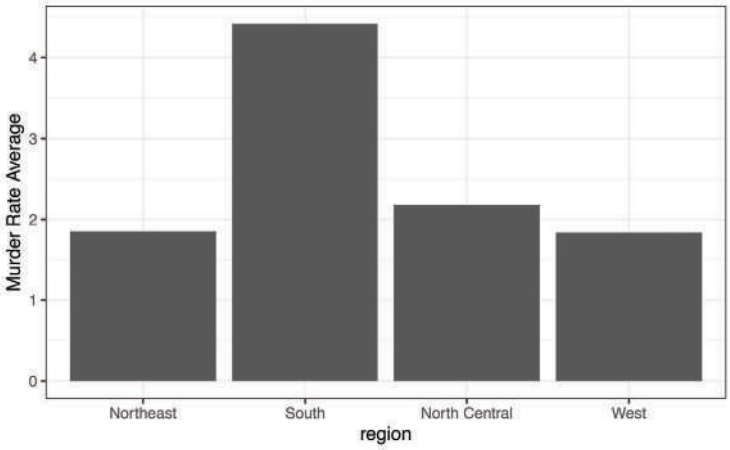
Redefine the `state` object so that the levels are re-ordered. Print the new object `state` and its levels so you can see that the vector is not re-ordered by the levels.

5. Now with one line of code, define the `dat` table as done above, but change the use mutate to create a rate variable and re-order the state variable so that the levels are re-ordered by this variable. Then make a barplot using the code above, but for this new `dat`.

6. Say we are interested in comparing gun homicide rates across regions of the US. We see this plot:

```
library(dslabs)
murders |> mutate(rate = total/population*100000) |>
group_by(region) |>
summarize(avg = mean(rate)) |>
mutate(region = factor(region)) |>
ggplot(aes(region, avg)) +
geom_col() +
ylab("Murder Rate Average")
```

ISTUDY



and decide to move to a state in the western region. What is the main problem with this interpretation?

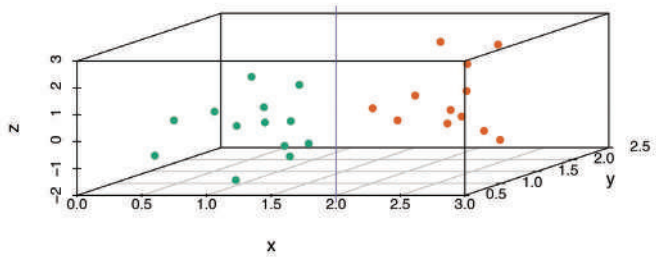
(a) The categories are ordered alphabetically.
(b) The graph does not show standard errors.
(c) It does not show all the data. We do not see the variability within a region and it's possible that the safest states are not in the West.
(d) The Northeast has the lowest average.

7. Make a boxplot of the murder rates defined as

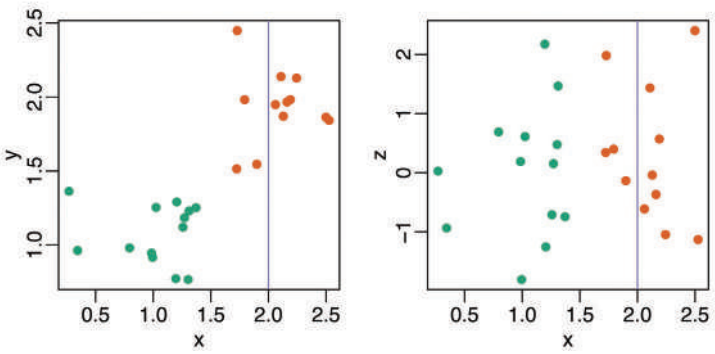
```
murders |> mutate(rate = total/population*100000)
```

by region, showing all the points and ordering the regions by their median rate.

8. The plots below show three continuous variables.



The line $x = 2$ appears to separate the points. But it is actually not the case, which we can see by plotting the data in a couple of two-dimensional points.



Why is this happening?

(a) Humans are not good at reading pseudo-3D plots.
(b) There must be an error in the code.
(c) The colors confuse us.
(d) Scatterplots should not be used to compare two variables when we have access to 3.

ISTUDY

10

Data visualization in practice

In this chapter, we will demonstrate how relatively simple **ggplot2** code can create insightful and aesthetically pleasing plots. As motivation we will create plots that help us better understand trends in world health and economics. We will implement what we learned in Chapters 8 and 9 and learn how to augment the code to perfect the plots. As we go through our case study, we will describe relevant general data visualization principles and learn concepts such as *faceting*, *time series plots*, *transformations*, and *ridge plots*.

10.1 Case study 1: new insights on poverty

Hans Rosling[1] was the co-founder of the Gapminder Foundation[2], an organization dedicated to educating the public by using data to dispel common myths about the so-called developing world. The organization uses data to show how actual trends in health and economics contradict the narratives that emanate from sensationalist media coverage of catastrophes, tragedies, and other unfortunate events. As stated in the Gapminder Foundation's website:

> Journalists and lobbyists tell dramatic stories. That's their job. They tell stories about extraordinary events and unusual people. The piles of dramatic stories pile up in peoples' minds into an over-dramatic world-view and strong negative stress feelings: "The world is getting worse!", "It's we vs. them!", "Other people are strange!", "The population just keeps growing!" and "Nobody cares!"

Hans Rosling conveyed actual data-based trends in a dramatic way of his own, using effective data visualization. This section is based on two talks that exemplify this approach to education: New Insights on Poverty[3] and The Best Stats You've Ever Seen[4]. Specifically, in this section, we use data to attempt to answer the following two questions:

1. Is it a fair characterization of today's world to say it is divided into western rich nations and the developing world in Africa, Asia, and Latin America?
2. Has income inequality across countries worsened during the last 40 years?

To answer these questions, we will be using the `gapminder` dataset provided in **dslabs**. This dataset was created using a number of spreadsheets available from the Gapminder Foundation. You can access the table like this:

[1] https://en.wikipedia.org/wiki/Hans_Rosling
[2] http://www.gapminder.org/
[3] https://www.ted.com/talks/hans_rosling_reveals_new_insights_on_poverty?language=en
[4] https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen

DOI: 10.1201/9781003220923-10

ISTUDY

```
library(tidyverse)
library(dslabs)
gapminder |> as_tibble()
#> # A tibble: 10,545 x 9
#>   country     year infant_mortality life_expectancy fertility population
#>   <fct>      <int>           <dbl>           <dbl>     <dbl>      <dbl>
#> 1 Albania     1960           115.             62.9      6.19    1636054
#> 2 Algeria     1960           148.             47.5      7.65   11124892
#> 3 Angola      1960           208              36.0      7.32    5270844
#> 4 Antigua a~  1960            NA              63.0      4.43      54681
#> 5 Argentina   1960            59.9            65.4      3.11   20619075
#> # i 10,540 more rows
#> # i 3 more variables: gdp <dbl>, continent <fct>, region <fct>
```

As done in the *New Insights on Poverty* video, we start by testing our knowledge regarding differences in child mortality across different countries. For each of the six pairs of countries below, which country do you think had the highest child mortality rates in 2015? Which pairs do you think are most similar?

1. Sri Lanka or Turkey
2. Poland or South Korea
3. Malaysia or Russia
4. Pakistan or Vietnam
5. Thailand or South Africa

When answering these questions without data, the non-European countries are typically picked as having higher child mortality rates: Sri Lanka over Turkey, South Korea over Poland, and Malaysia over Russia. It is also common to assume that countries considered to be part of the developing world: Pakistan, Vietnam, Thailand, and South Africa, have similarly high mortality rates.

To answer these questions **with data**, we can use **dplyr**. For example, for the first comparison we see that:

```
gapminder |>
  filter(year == 2015 & country %in% c("Sri Lanka","Turkey")) |>
  select(country, infant_mortality)
#>     country infant_mortality
#> 1 Sri Lanka              8.4
#> 2    Turkey             11.6
```

Turkey has the higher infant mortality rate.

We can use this code on all comparisons and find the following:

country	infant_mortality	country	infant_mortality
Sri Lanka	8.4	Turkey	11.6
Poland	4.5	South Korea	2.9
Malaysia	6.0	Russia	8.2
Pakistan	65.8	Vietnam	17.3
Thailand	10.5	South Africa	33.6

ISTUDY

We see that the European countries on this list have higher child mortality rates: Poland has a higher rate than South Korea, and Russia has a higher rate than Malaysia. We also see that Pakistan has a much higher rate than Vietnam, and South Africa has a much higher rate than Thailand. It turns out that when Hans Rosling gave this quiz to educated groups of people, the average score was less than 2.5 out of 5, worse than what they would have obtained had they guessed randomly. This implies that more than ignorant, we are misinformed. In this chapter we see how data visualization helps inform us.

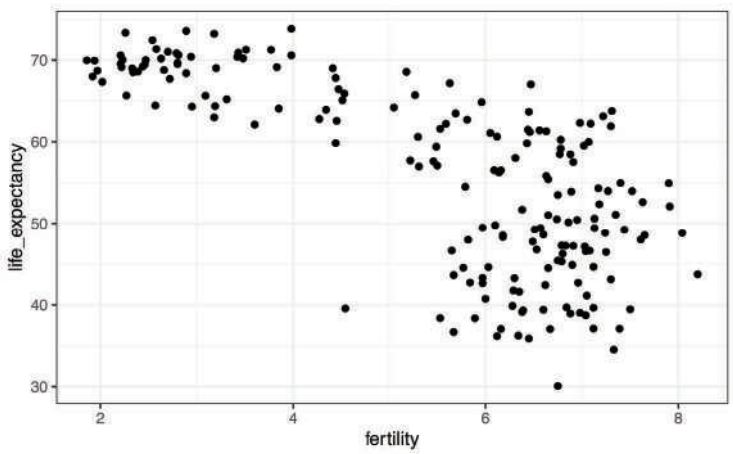
10.2 Scatterplots

The reason for the misconception described in the previous section stems from the preconceived notion that the world is divided into two groups: the western world (Western Europe and North America), characterized by long life spans and small families, versus the developing world (Africa, Asia, and Latin America) characterized by short life spans and large families. But do the data support this dichotomous view?

The necessary data to answer this question is also available in our `gapminder` table. Using our newly learned data visualization skills, we will be able to tackle this challenge.

In order to analyze this world view, our first plot is a scatterplot of life expectancy versus fertility rates (average number of children per woman). We start by looking at data from about 50 years ago, when perhaps this view was first cemented in our minds.

```
filter(gapminder, year == 1962) |>
  ggplot(aes(fertility, life_expectancy)) +
  geom_point()
```

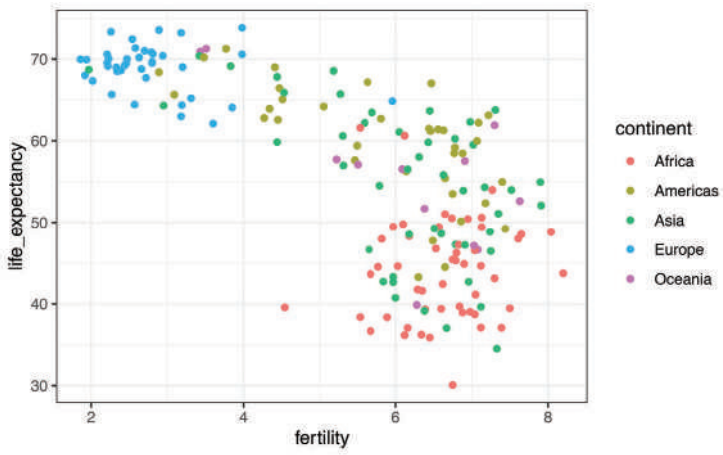


Most points fall into two distinct categories:

1. Life expectancy around 70 years and 3 or fewer children per family.
2. Life expectancy lower than 65 years and more than 5 children per family.

To confirm that indeed these countries are from the regions we expect, we can use color to represent continent.

```
filter(gapminder, year == 1962) |>
  ggplot( aes(fertility, life_expectancy, color = continent)) +
  geom_point()
```



In 1962, "the West versus developing world" view was grounded in some reality. Is this still the case 50 years later?

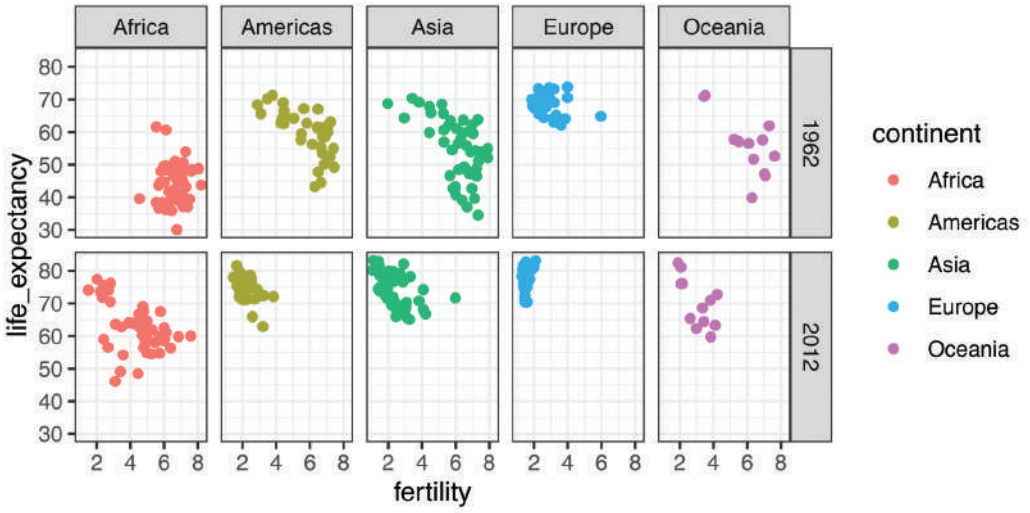
10.3 Faceting

We could easily plot the 2012 data in the same way we did for 1962. To make comparisons, however, side by side plots are preferable. In **ggplot2**, we can achieve this by *faceting* variables: we stratify the data by some variable and make the same plot for each strata.

To achieve faceting, we add a layer with the function `facet_grid`, which automatically separates the plots. This function lets you facet by up to two variables using columns to represent one variable and rows to represent the other. The function expects the row and column variables to be separated by a `~`. Here is an example of a scatterplot with `facet_grid` added as the last layer:

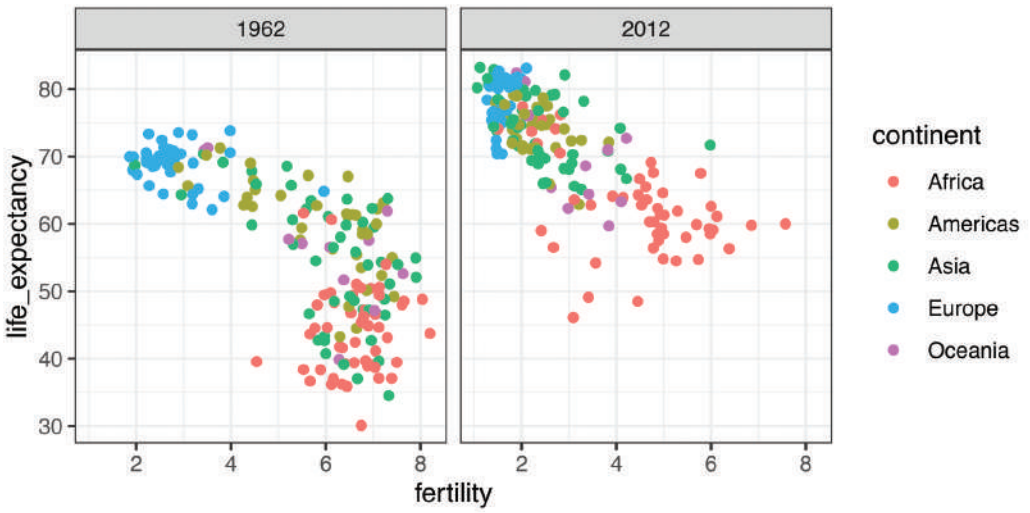
```
filter(gapminder, year %in% c(1962, 2012)) |>
  ggplot(aes(fertility, life_expectancy, col = continent)) +
  geom_point() +
  facet_grid(year~continent)
```

ISTUDY



We see a plot for each continent/year pair. However, this is just an example and more than what we want, which is simply to compare 1962 and 2012. In this case, there is just one variable and we use `.` to let facet know that we are not using a second variable:

```
filter(gapminder, year %in% c(1962, 2012)) |>
  ggplot(aes(fertility, life_expectancy, col = continent)) +
  geom_point() +
  facet_grid(. ~ year)
```

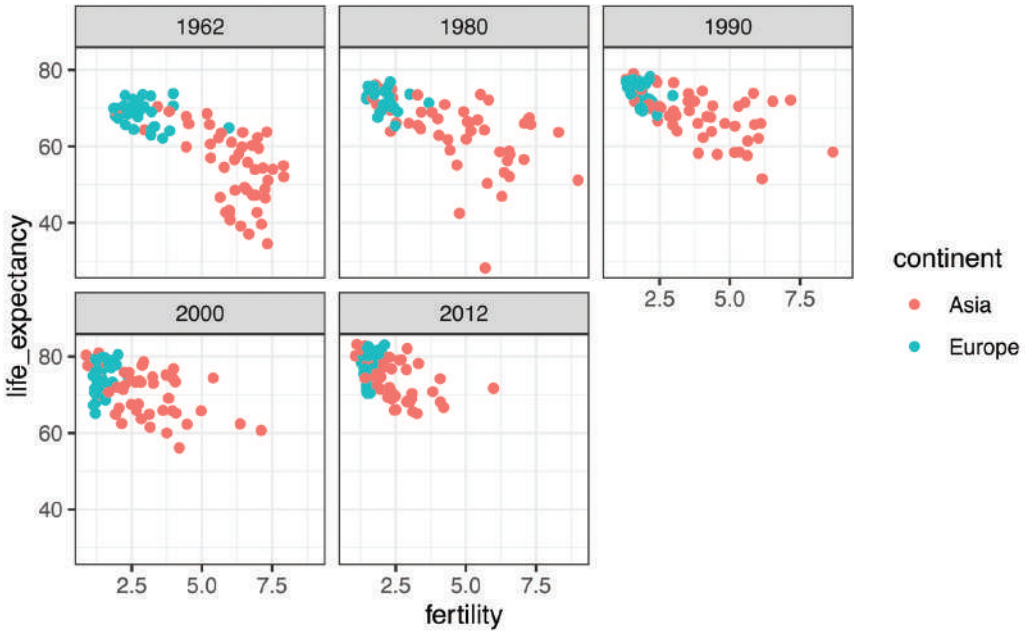


This plot clearly shows that the majority of countries have moved from the *developing world* cluster to the *western world* one. In 2012, the western versus developing world view no longer makes sense. This is particularly clear when comparing Europe to Asia, the latter of which includes several countries that have made great improvements.

10.3.1 `facet_wrap`

To explore how this transformation happened through the years, we can make the plot for several years. For example, we can add 1970, 1980, 1990, and 2000. If we do this, we will not want all the plots on the same row, the default behavior of `facet_grid`, since they will become too thin to show the data. Instead, we will want to use multiple rows and columns. The function `facet_wrap` permits us to do this by automatically wrapping the series of plots so that each display has viewable dimensions:

```
years <- c(1962, 1980, 1990, 2000, 2012)
continents <- c("Europe", "Asia")
gapminder |>
  filter(year %in% years & continent %in% continents) |>
  ggplot( aes(fertility, life_expectancy, col = continent)) +
  geom_point() +
  facet_wrap(~year)
```



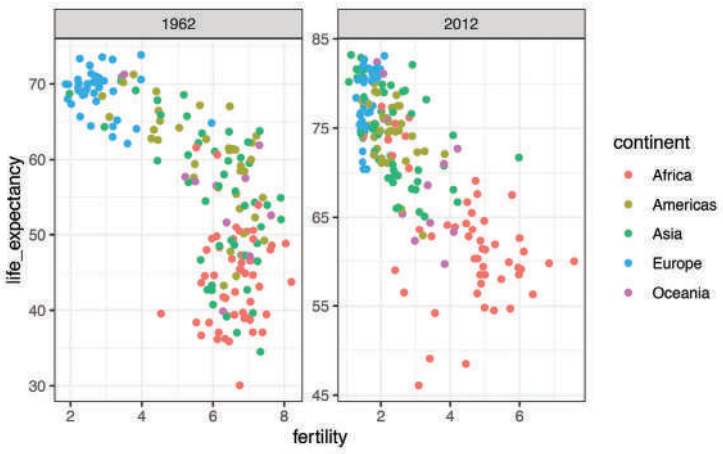
This plot clearly shows how most Asian countries have improved at a much faster rate than European ones.

10.3.2 Fixed scales for better comparisons

The default choice of the range of the axes is important. When not using `facet`, this range is determined by the data shown in the plot. When using `facet`, this range is determined by the data shown in all plots and therefore kept fixed across plots. This makes comparisons

across plots much easier. For example, in the above plot, we can see that life expectancy has increased and the fertility has decreased across most countries. We see this because the cloud of points moves. This is not the case if we adjust the scales:

```
filter(gapminder, year %in% c(1962, 2012)) |>
  ggplot(aes(fertility, life_expectancy, col = continent)) +
  geom_point() +
  facet_wrap(. ~ year, scales = "free")
```



In the plot above, we have to pay special attention to the range to notice that the plot on the right has a larger life expectancy.

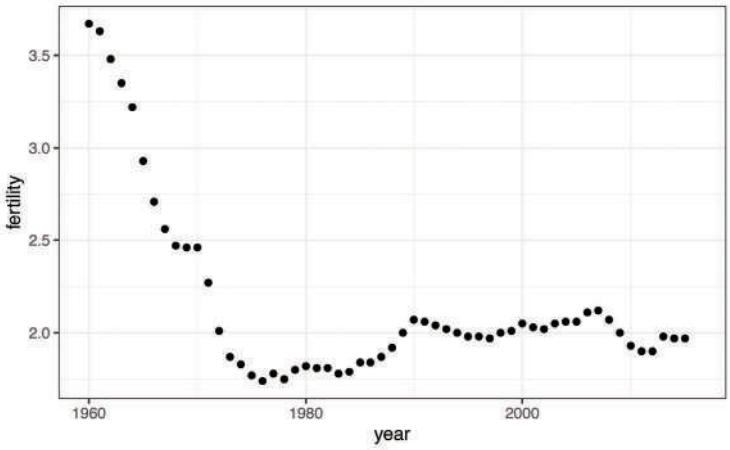
10.4 Time series plots

The visualizations above effectively illustrate that data no longer supports the western versus developing world view. Once we see these plots, new questions emerge. For example, which countries are improving more and which ones less? Was the improvement constant during the last 50 years or was it more accelerated during certain periods? For a closer look that may help answer these questions, we introduce *time series plots*.

Time series plots have time in the x-axis and an outcome or measurement of interest on the y-axis. For example, here is a trend plot of United States fertility rates:

```
gapminder |>
  filter(country == "United States") |>
  ggplot(aes(year, fertility)) +
  geom_point()
```

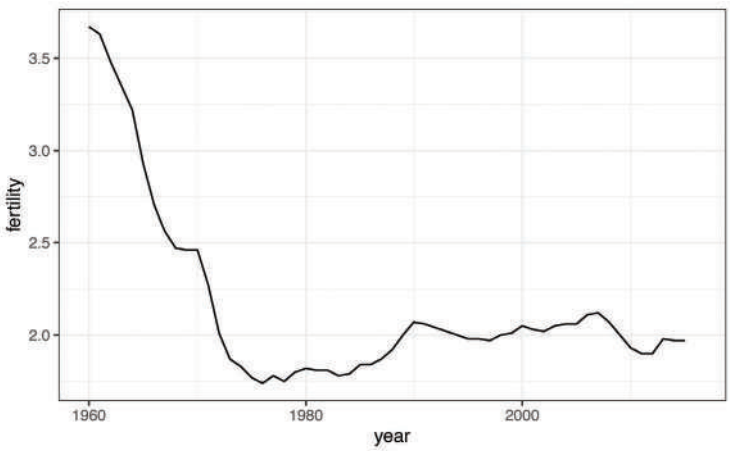
ISTUDY



We see that the trend is not linear at all. Instead there is sharp drop during the 1960s and 1970s to below 2. Then the trend comes back to 2 and stabilizes during the 1990s.

When the points are regularly and densely spaced, as they are here, we create curves by joining the points with lines, to convey that these data are from a single series, here a country. To do this, we use the `geom_line` function instead of `geom_point`.

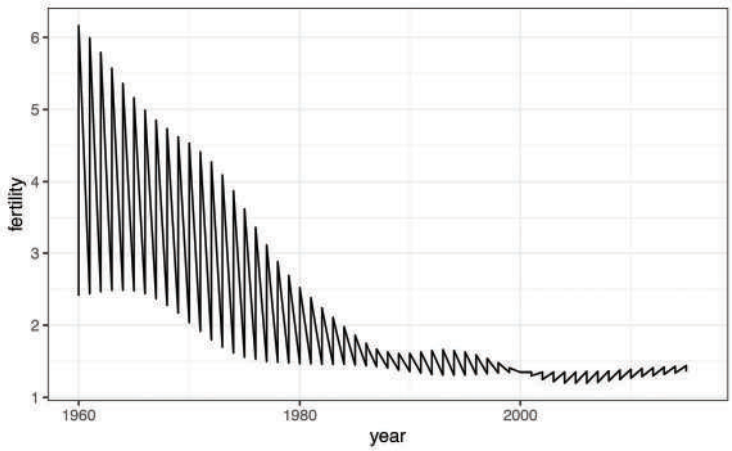
```
gapminder |>
  filter(country == "United States") |>
  ggplot(aes(year, fertility)) +
  geom_line()
```



This is particularly helpful when we look at two countries. We can subset the data to include two countries, one from Europe and one from Asia, then adapt the code above:

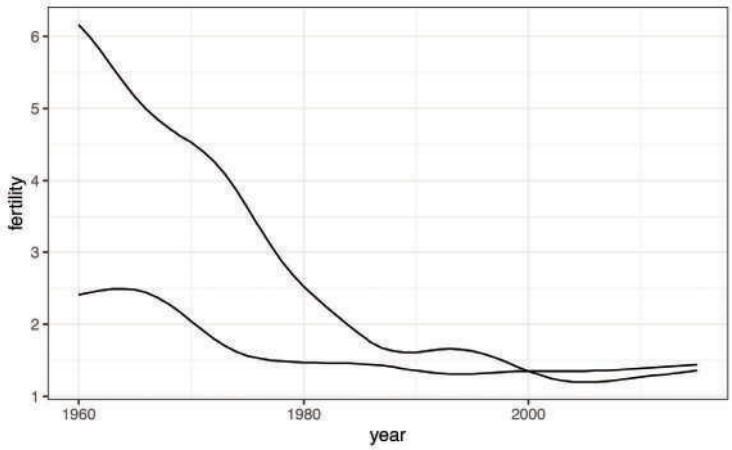
ISTUDY

```
countries <- c("South Korea", "Germany")

gapminder |> filter(country %in% countries) |>
  ggplot(aes(year,fertility)) +
  geom_line()
```



Unfortunately, this is **not** the plot that we want. Rather than a line for each country, the points for both countries are joined. This is actually expected since we have not told `ggplot` anything about wanting two separate lines. To let `ggplot` know that there are two curves that need to be made separately, we assign each point to a `group`, one for each country:

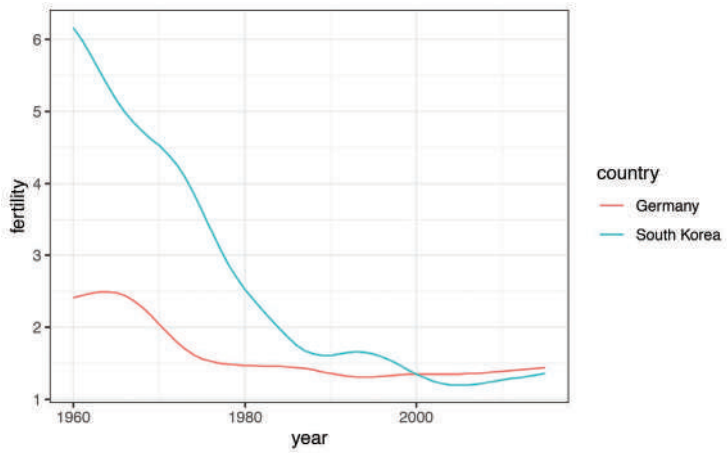
```
countries <- c("South Korea","Germany")

gapminder |> filter(country %in% countries & !is.na(fertility)) |>
  ggplot(aes(year, fertility, group = country)) +
  geom_line()
```



ISTUDY

But which line goes with which country? We can assign colors to make this distinction. A useful side-effect of using the `color` argument to assign different colors to the different countries is that the data is automatically grouped:

```
countries <- c("South Korea","Germany")
gapminder |> filter(country %in% countries & !is.na(fertility)) |>
  ggplot(aes(year,fertility, col = country)) +
  geom_line()
```



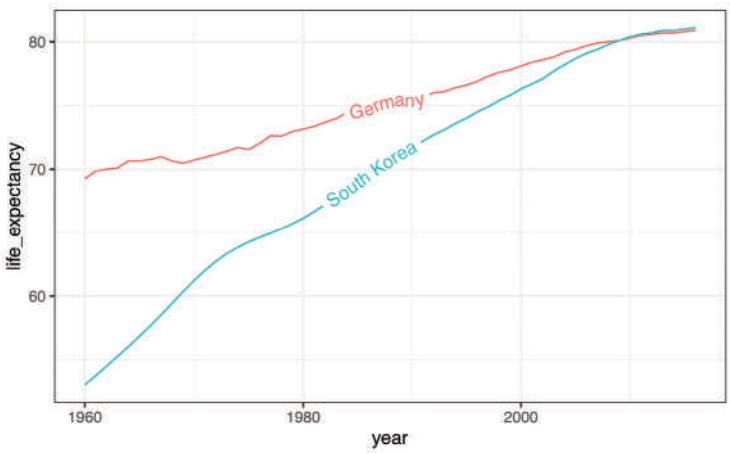
The plot clearly shows how South Korea's fertility rate dropped drastically during the 1960s and 1970s, and by 1990 had a similar rate to that of Germany.

For trend plots we recommend labeling the lines rather than using legends since the viewer can quickly see which line is which country. This suggestion actually applies to most plots: labeling is usually preferred over legends.

We demonstrate how we can do this using the `geomtextpath` package. We define a data table with the label locations and then use a second mapping just for these labels:

```
library(geomtextpath)
gapminder |>
  filter(country %in% countries) |>
  ggplot(aes(year, life_expectancy, col = country, label = country)) +
  geom_textpath() +
  theme(legend.position = "none")
```

ISTUDY



The plot clearly shows how an improvement in life expectancy followed the drops in fertility rates. In 1960, Germans lived 15 years longer than South Koreans, although by 2010 the gap is completely closed. It exemplifies the improvement that many non-western countries have achieved in the last 40 years.

10.5 Data transformations

We now shift our attention to the second question related to the commonly held notion that wealth distribution across the world has become worse during the last decades. When general audiences are asked if poor countries have become poorer and rich countries become richer, the majority answers yes. By using stratification, histograms, smooth densities, and boxplots, we will be able to understand if this is in fact the case. First we learn how transformations can sometimes help provide more informative summaries and plots.

The `gapminder` data table includes a column with the countries' gross domestic product (GDP). GDP measures the market value of goods and services produced by a country in a year. The GDP per person is often used as a rough summary of a country's wealth. Here we divide this quantity by 365 to obtain the more interpretable measure *dollars per day*. Using current US dollars as a unit, a person surviving on an income of less than $2 a day is defined to be living in *absolute poverty*. We add this variable to the data table:

```
gapminder <- gapminder |>
  mutate(dollars_per_day = gdp/population/365)
```

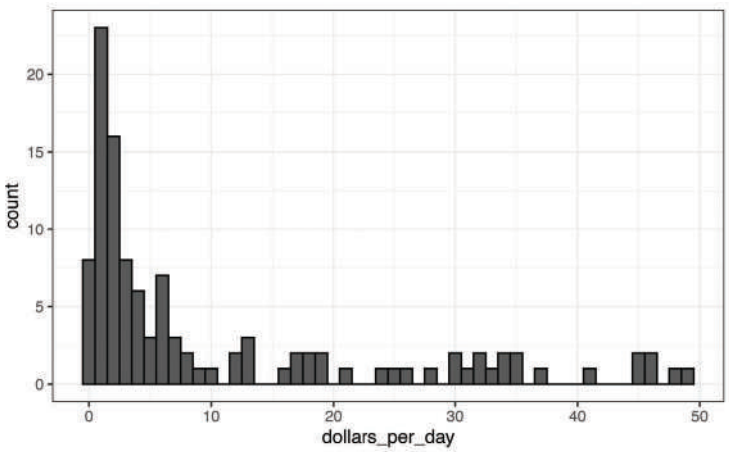
The GDP values are adjusted for inflation and represent current US dollars, so these values are meant to be comparable across the years. Of course, these are country averages and within each country there is much variability. All the graphs and insights described below relate to country averages and not to individuals.

ISTUDY

10.5.1 Log transformation

Here is a histogram of per day incomes from 1970:

```
past_year <- 1970
gapminder |>
  filter(year == past_year & !is.na(gdp)) |>
  ggplot(aes(dollars_per_day)) +
  geom_histogram(binwidth = 1, color = "black")
```



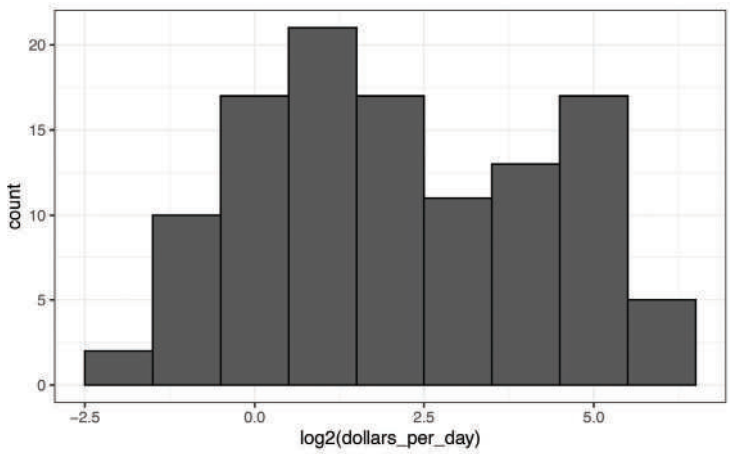
We use the `color = "black"` argument to draw a boundary and clearly distinguish the bins.

In this plot, we see that for the majority of countries, averages are below $10 a day. However, the majority of the x-axis is dedicated to the 35 countries with averages above $10. So the plot is not very informative about countries with values below $10 a day.

It might be more informative to quickly be able to see how many countries have average daily incomes of about $1 (extremely poor), $2 (very poor), $4 (poor), $8 (middle), $16 (well off), $32 (rich), and $64 (very rich) per day. These changes are multiplicative, and log transformations convert multiplicative changes into additive ones: when using base 2, a doubling of a value turns into an increase by 1.

Here is the distribution if we apply a log base 2 transform:

```
gapminder |>
  filter(year == past_year & !is.na(gdp)) |>
  ggplot(aes(log2(dollars_per_day))) +
  geom_histogram(binwidth = 1, color = "black")
```



In a way, this provides a *close-up* of the mid to lower income countries.

10.5.2 Which base?

In the case above, we used base 2 in the log transformations. Other common choices are base e (the natural log) and base 10.

In general, we do not recommend using the natural log for data exploration and visualization. This is because while $2^2, 2^3, 2^4, \ldots$, or $10^2, 10^3, \ldots$, are easy to mentally compute, but the same is not true for $e^2, e^3, \ldots$. The natural log scale is not intuitive or easy to interpret.

In the dollars per day example, we used base 2 instead of base 10 because the resulting range is easier to interpret. The range of the untransformed values is 0.3269426, 48.8852142.

In base 10, this turns into a range that includes very few integers: just 0 and 1. With base 2, our range includes -2, -1, 0, 1, 2, 3, 4, and 5. It is easier to compute 2^x and 10^x when x is an integer and between -10 and 10, so we prefer to have smaller integers in the scale. Another consequence of a limited range is that choosing the binwidth is more challenging. With log base 2, we know that a binwidth of 1 will translate to a bin with range x to $2x$.

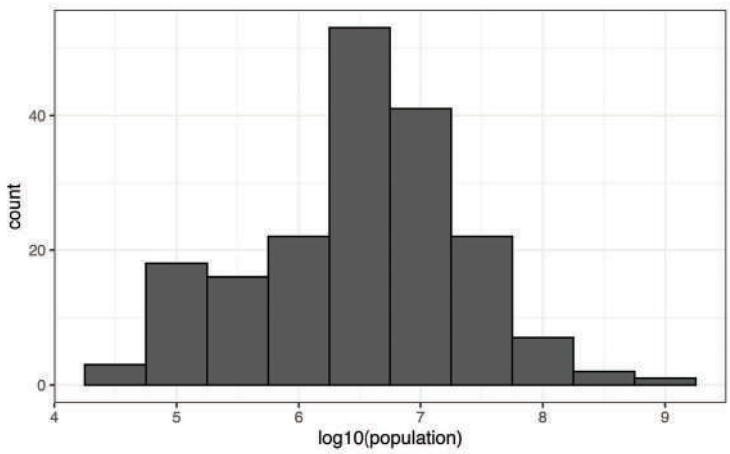
For an example in which base 10 makes more sense, consider population sizes. A log base 10 is preferable since the range for these is:

```
filter(gapminder, year == past_year) |>
  summarize(min = min(population), max = max(population))
#>     min      max
#> 1 46075 8.09e+08
```

Here is the histogram of the transformed values:

```
gapminder |>
  filter(year == past_year) |>
  ggplot(aes(log10(population))) +
  geom_histogram(binwidth = 0.5, color = "black")
```

ISTUDY



In the above, we quickly see that country populations range between ten thousand and ten billion.

10.5.3 Transform the values or the scale?

There are two ways we can use log transformations in plots. We can log the values before plotting them or use log scales in the axes. The plot will look the same, except for the numbers in the axes. Both approaches are useful and have different strengths. If we log the data, we can more easily interpret intermediate values in the scale. For example, if we see:

```
----1----x----2--------3----
```

for log transformed data, we know that the value of x is about 1.5. If the scales are logged:

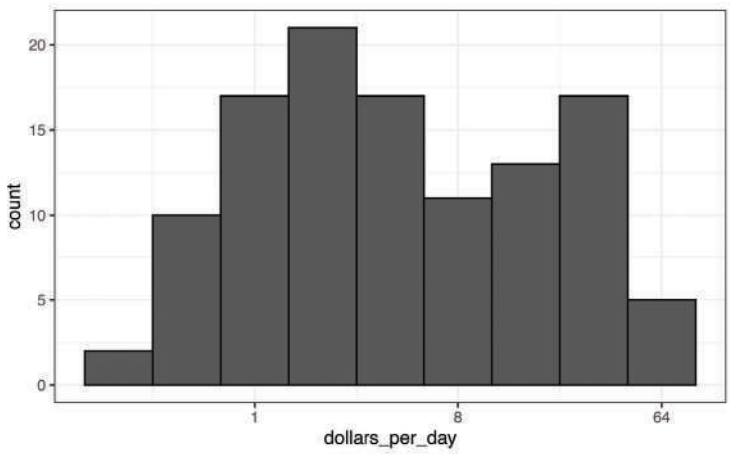
```
----10---x---100------1000---
```

then, to determine `x`, we need to compute $10^{1.5}$, which is not easy to do in our heads. However, the advantage of showing logged scales is that the original values are displayed in the plot, which are easier to interpret. For example, we would see "32 dollars a day" instead of "5 log base 2 dollars a day".

As we learned earlier, if we want to scale the axis with logs, we can use the `scale_x_continuous` function. Instead of logging the values first, we apply this layer:

```
gapminder |>
  filter(year == past_year & !is.na(gdp)) |>
  ggplot(aes(dollars_per_day)) +
  geom_histogram(binwidth = 1, color = "black") +
  scale_x_continuous(trans = "log2")
```

ISTUDY



Note that the log base 10 transformation has its own function: `scale_x_log10()`, but currently base 2 does not, although we could easily define our own.

There are other transformations available through the `trans` argument. As we learn later on, the square root (`sqrt`) transformation is useful when considering counts. The logistic transformation (`logit`) is useful when plotting proportions between 0 and 1. The `reverse` transformation is useful when we want smaller values to be on the right or on top.

10.6 Multimodal distributions

In the histogram above we see two *bumps*: one at about 4 and another at about 32. In statistics these bumps are sometimes referred to as *modes*. The mode of a distribution is the value with the highest frequency. The mode of the normal distribution is the average. When a distribution, like the one above, doesn't monotonically decrease from the mode, we call the locations where it goes up and down again *local modes* and say that the distribution has *multiple modes*.

The histogram above suggests that the 1970 country income distribution has two modes: one at about 2 dollars per day (1 in the log 2 scale) and another at about 32 dollars per day (5 in the log 2 scale). This *bimodality* is consistent with a dichotomous world made up of countries with average incomes less than 8 dollars per day (3 in the log 2 scale) and countries above that.

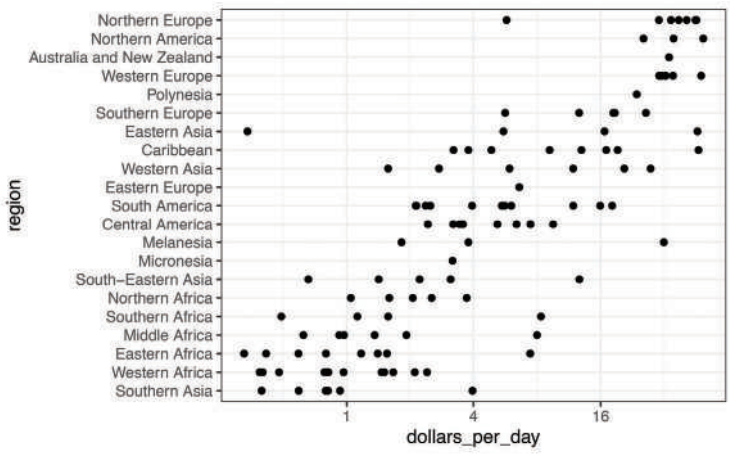
10.7 Comparing distributions

A histogram showed us that the 1970 income distribution values show a dichotomy. However, the histogram does not show us if the two groups of countries are *west* versus the *developing* world.

Let's start by quickly examining the data by region. We reorder the regions by the median value and use a log scale.

ISTUDY

```
gapminder |>
  filter(year == past_year & !is.na(gdp)) |>
  mutate(region = reorder(region, dollars_per_day, FUN = median)) |>
  ggplot(aes(dollars_per_day, region)) +
  geom_point() +
  scale_x_continuous(trans = "log2")
```



We can already see that there is indeed a "west versus the rest" dichotomy: we see two clear groups, with the rich group composed of North America, Northern and Western Europe, and New Zealand and Australia. We define groups based on this observation:

```
gapminder <- gapminder |>
  mutate(group = case_when(
    region %in% c("Western Europe", "Northern Europe","Southern Europe",
                  "Northern America",
                  "Australia and New Zealand") ~ "West",
    region %in% c("Eastern Asia", "South-Eastern Asia") ~ "East Asia",
    region %in% c("Caribbean", "Central America",
                  "South America") ~ "Latin America",
    continent == "Africa" &
      region != "Northern Africa" ~ "Sub-Saharan",
    TRUE ~ "Others"))
```

We turn this `group` variable into a factor to control the order of the levels:

```
gapminder <- gapminder |>
  mutate(group = factor(group, levels = c("Others", "Latin America",
                                          "East Asia", "Sub-Saharan",
                                          "West")))
```

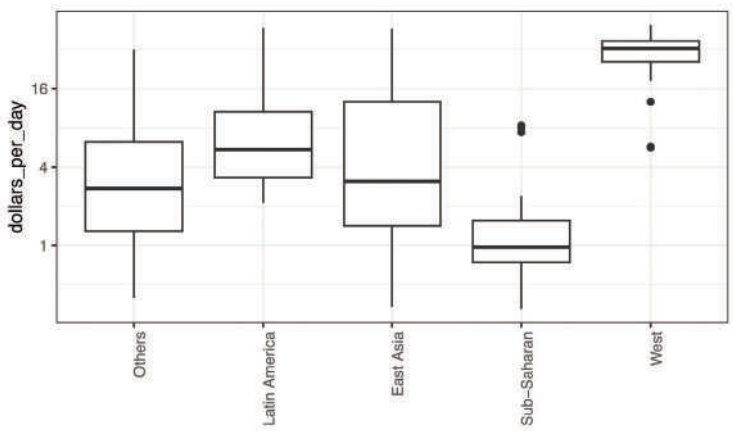
In the next section we demonstrate how to visualize and compare distributions across groups.

ISTUDY

10.7.1 Boxplots

The exploratory data analysis above has revealed two characteristics about average income distribution in 1970. Using a histogram, we found a bimodal distribution with the modes relating to poor and rich countries. We now want to compare the distribution across these five groups to confirm the "west versus the rest" dichotomy. The number of points in each category is large enough that a summary plot may be useful. We could generate five histograms or five density plots, but it may be more practical to have all the visual summaries in one plot. We therefore start by stacking boxplots next to each other. Note that we add the layer `theme(axis.text.x = element_text(angle = 90, hjust = 1))` to turn the group labels vertical, since they do not fit if we show them horizontally, and we remove the axis label to make space.

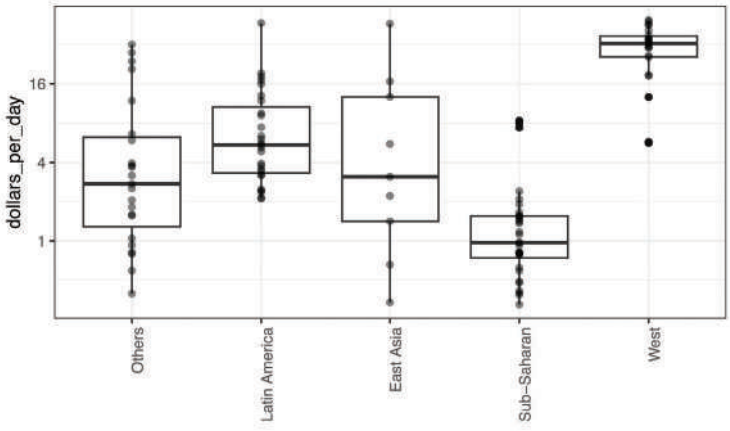
```
p <- gapminder |>
  filter(year == past_year & !is.na(gdp)) |>
  ggplot(aes(group, dollars_per_day)) +
  geom_boxplot() +
  scale_y_continuous(trans = "log2") +
  xlab("") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
p
```



Boxplots have the limitation that by summarizing the data into five numbers, we might miss important characteristics of the data. One way to avoid this is by showing the data.

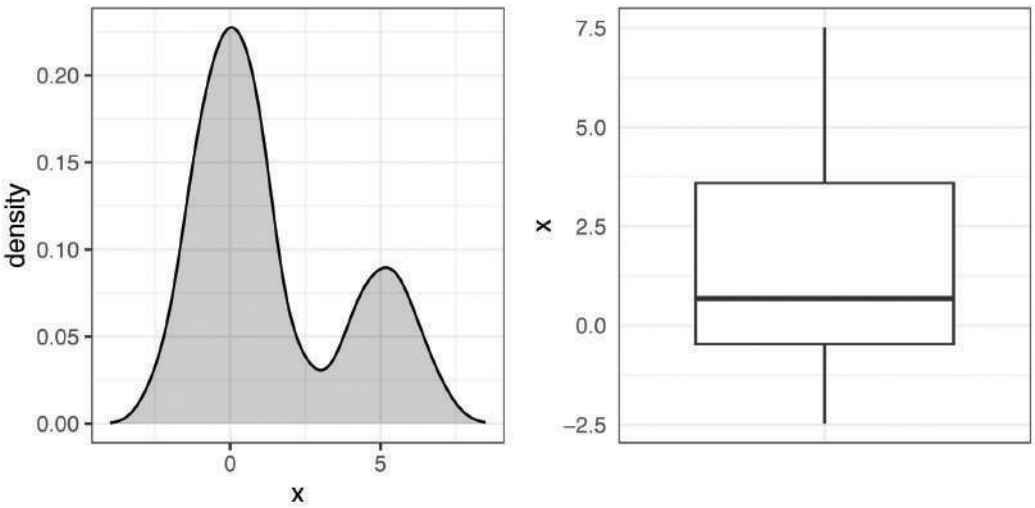
```
p + geom_point(alpha = 0.5)
```

ISTUDY



10.7.2 Ridge plots

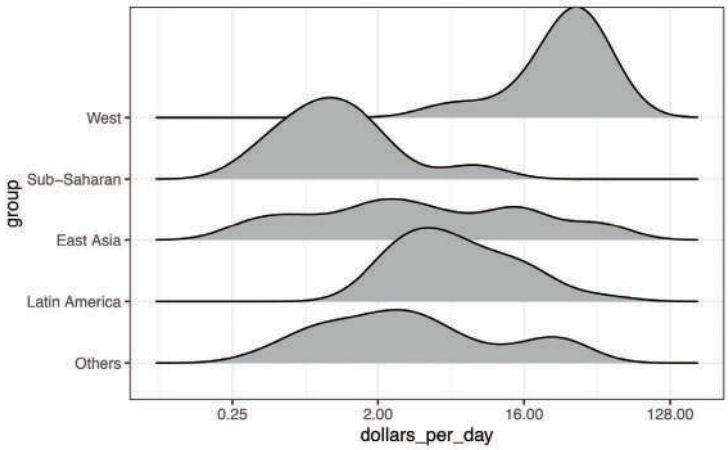
Showing each individual point does not always reveal important characteristics of the distribution. Although not the case here, when the number of data points is so large that there is over-plotting, showing the data can be counterproductive. Boxplots help with this by providing a five-number summary, but this has limitations too. For example, boxplots will not permit us to discover bimodal distributions. To see this, note that the two plots below are summarizing the same dataset:



In cases in which we are concerned that the boxplot summary is too simplistic, we can show stacked smooth densities or histograms. We refer to these as *ridge plots*. Because we are used to visualizing densities with values in the x-axis, we stack them vertically. Also, because more space is needed in this approach, it is convenient to overlay them. The package **ggridges** provides a convenient function for doing this. Here is the income data shown above with boxplots but with a *ridge plot*.

ISTUDY

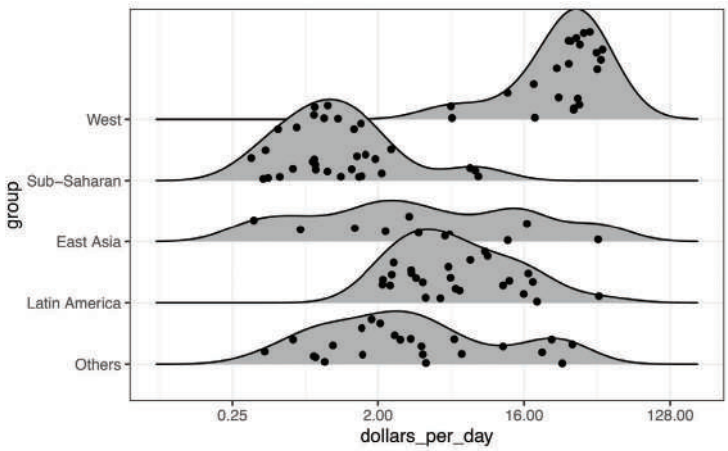
```
library(ggridges)
p <- gapminder |>
  filter(year == past_year & !is.na(dollars_per_day)) |>
  ggplot(aes(dollars_per_day, group)) +
  scale_x_continuous(trans = "log2")
p  + geom_density_ridges()
```



Note that we have to invert the `x` and `y` used for the boxplot. A useful `geom_density_ridges` parameter is `scale`, which lets you determine the amount of overlap, with `scale = 1` meaning no overlap and larger values resulting in more overlap.

If the number of data points is small enough, we can add them to the ridge plot using the following code:

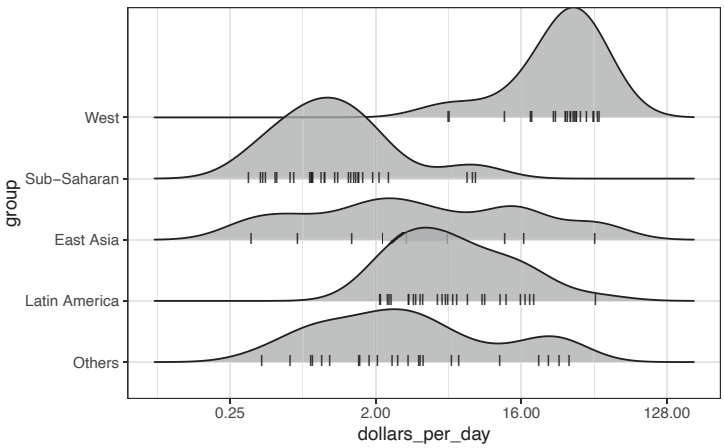
```
p + geom_density_ridges(jittered_points = TRUE)
```



ISTUDY

By default, the height of the points is jittered and should not be interpreted in any way. To show data points, but without using jitter we can use the following code to add what is referred to as a *rug representation* of the data.

```
p + geom_density_ridges(jittered_points = TRUE,
                        position = position_points_jitter(height = 0),
                        point_shape = '|', point_size = 3,
                        point_alpha = 1, alpha = 0.7)
```



10.7.3 Example: 1970 versus 2010 income distributions

Data exploration clearly shows that in 1970 there was a "west versus the rest" dichotomy. But does this dichotomy persist? Let's use `facet_grid` and see how the distributions have changed. To start, we will focus on two groups: the west and the rest. We make four histograms. We make this plot only for countries with data in both 1970 and 2010. Note that several countries were founded after 1970; for example, the Soviet Union divided into several countries during the 1990s. We also note that that data was available for more countries in 2010.

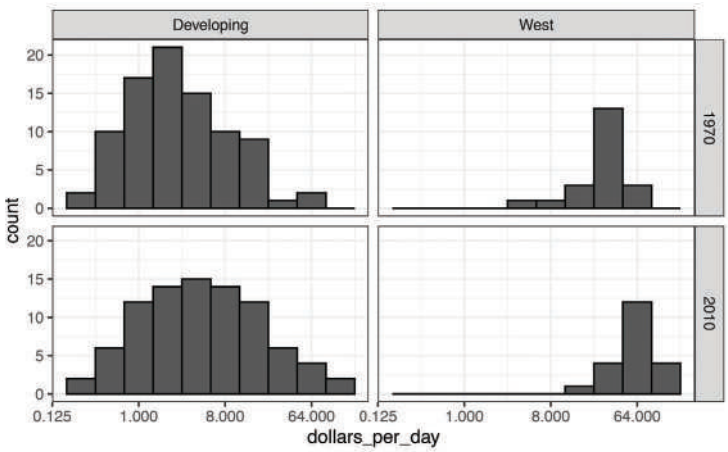
We therefore make the plot only for countries with data in both years:

```
past_year <- 1970
present_year <- 2010
years <- c(past_year, present_year)
country_list <- gapminder |>
  filter(year %in% c(present_year, past_year)) |>
  group_by(country) |>
  summarize(n = sum(!is.na(dollars_per_day)), .groups = "drop") |>
  filter(n == 2) |>
  pull(country)
```

ISTUDY

These 108 countries account for 86% of the world population, so this subset should be representative. We can compare the distributions using this code:

```
gapminder |>
  filter(year %in% years & country %in% country_list) |>
  mutate(west = ifelse(group == "West", "West", "Developing")) |>
  ggplot(aes(dollars_per_day)) +
  geom_histogram(binwidth = 1, color = "black") +
  scale_x_continuous(trans = "log2") +
  facet_grid(year ~ west)
```

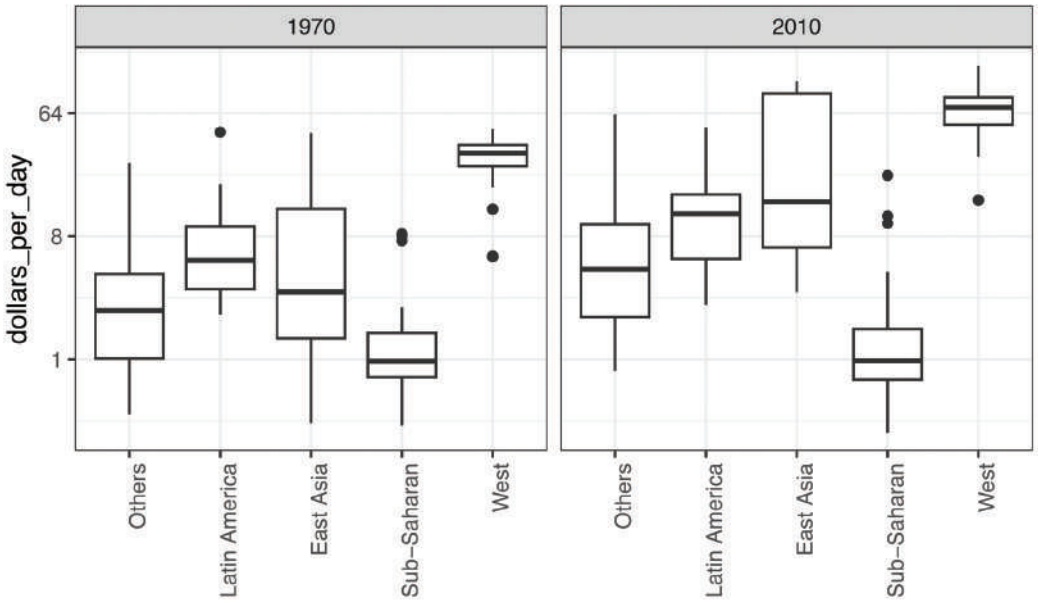


We now see that the rich countries have become a bit richer, but percentage-wise, the poor countries appear to have improved more. In particular, we see that the proportion of *developing* countries earning more than $16 a day increased substantially.

To see which specific regions improved the most, we can remake the boxplots we made above, but now adding the year 2010 and then using facet to compare the two years.

```
gapminder |>
  filter(year %in% years & country %in% country_list) |>
  ggplot(aes(group, dollars_per_day)) +
  geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  scale_y_continuous(trans = "log2") +
  xlab("") +
  facet_grid(. ~ year)
```

ISTUDY

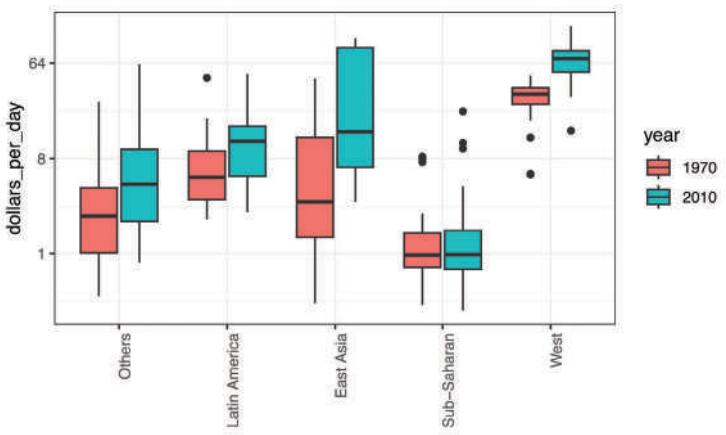


Here, we pause to introduce another powerful **ggplot2** feature. Because we want to compare each region before and after, it would be convenient to have the 1970 boxplot next to the 2010 boxplot for each region. In general, comparisons are easier when data are plotted next to each other.

So instead of faceting, we keep the data from each year together and ask to color (or fill) them depending on the year. Note that groups are automatically separated by year and each pair of boxplots drawn next to each other. Because the year column is a number, we have to convert it into a factor since **ggplot2** automatically assigns a color to each category of a factor.

```
gapminder |>
  filter(year %in% years & country %in% country_list) |>
  mutate(year = factor(year)) |>
  ggplot(aes(group, dollars_per_day, fill = year)) +
  geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  scale_y_continuous(trans = "log2") +
  xlab("")
```

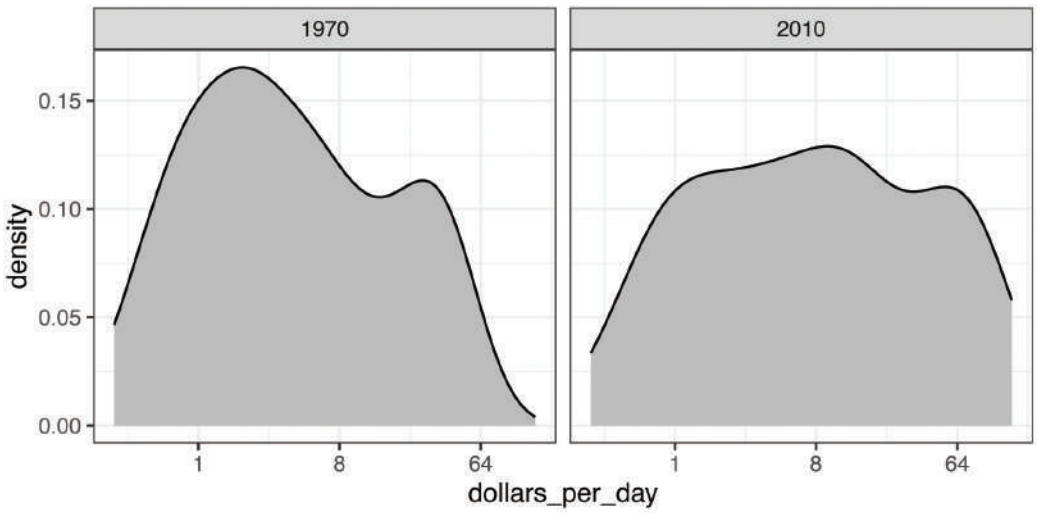
ISTUDY



The previous data exploration suggested that the income gap between rich and poor countries has narrowed considerably during the last 40 years. We used a series of histograms and boxplots to see this. We suggest a succinct way to convey this message with just one plot.

Let's start by noting that density plots for income distribution in 1970 and 2010 deliver the message that the gap is closing:

```
gapminder |>
  filter(year %in% years & country %in% country_list) |>
  ggplot(aes(dollars_per_day)) +
  geom_density(fill = "grey") +
  scale_x_continuous(trans = "log2") +
  facet_grid(. ~ year)
```



In the 1970 plot, we see two clear modes: poor and rich countries. In 2010, it appears that some of the poor countries have shifted toward the right, closing the gap.

ISTUDY

The next message we need to convey is that the reason for this change in distribution is that several poor countries became richer, rather than some rich countries becoming poorer. To do this, we can assign a color to the groups we identified during data exploration.

However, because when we overlay two densities, the default is to have the area under the distribution curve add up to 1 for each group, regardless of the size of each group, we first need to learn how to make these smooth densities in a way that preserves information on the number of countries in each group. To do this, we will need to learn to access computed variables with the `geom_density` function.

10.7.4 Accessing computed variables

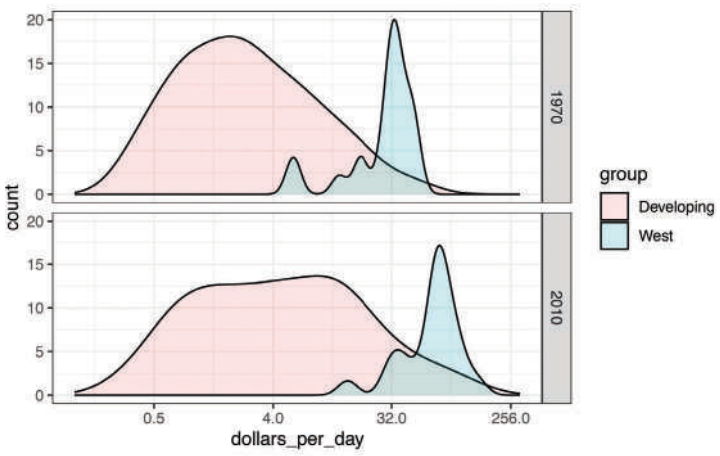
To have the areas of these densities be proportional to the size of the groups, we can simply multiply the y-axis values by the size of the group. From the `geom_density` help file, we see that the functions compute a variable called `count` that does exactly this. We want this variable to be on the y-axis rather than the density.

In **ggplot2**, we access these variables using the function `after_stat`. We will therefore use the following mapping:

```
aes(x = dollars_per_day, y = after_stat(count))
```

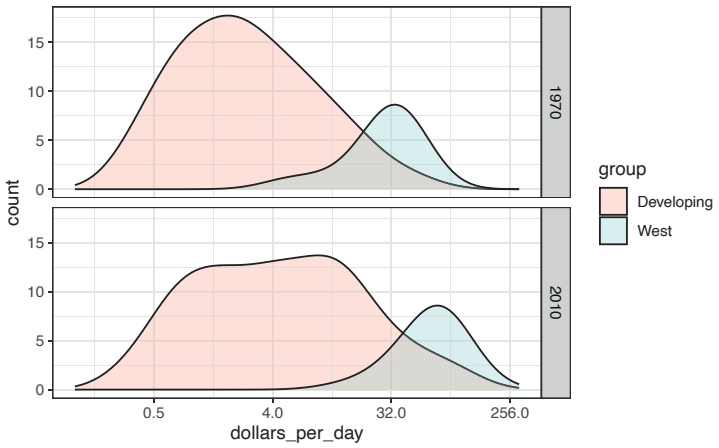
We can now create the desired plot by simply changing the mapping in the previous code chunk. We will also expand the limits of the x-axis.

```
p <- gapminder |>
  filter(year %in% years & country %in% country_list) |>
  mutate(group = ifelse(group == "West", "West", "Developing")) |>
  ggplot(aes(dollars_per_day, y = after_stat(count), fill = group)) +
  scale_x_continuous(trans = "log2", limits = c(0.125, 300))
p + geom_density(alpha = 0.2) + facet_grid(year ~ .)
```



If we want the densities to be smoother, we use the `bw` argument so that the same bandwidth is used in each density. We selected 0.75 after trying out several values.

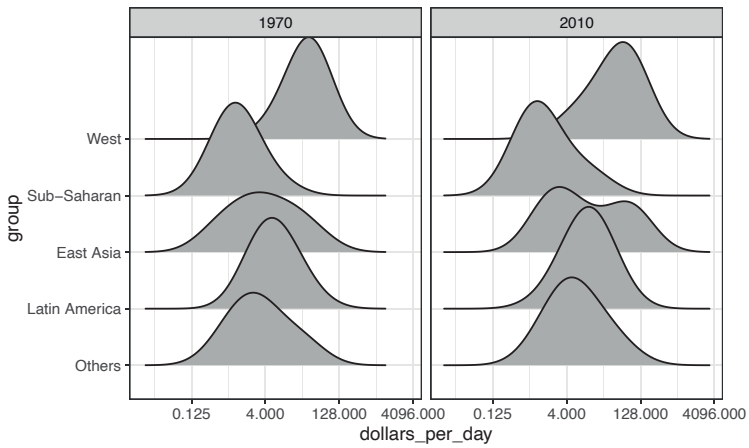
```
p + geom_density(alpha = 0.2, bw = 0.75) + facet_grid(year ~ .)
```



This plot now shows what is happening very clearly. The developing world distribution is changing. A third mode appears consisting of the countries that most narrowed the gap.

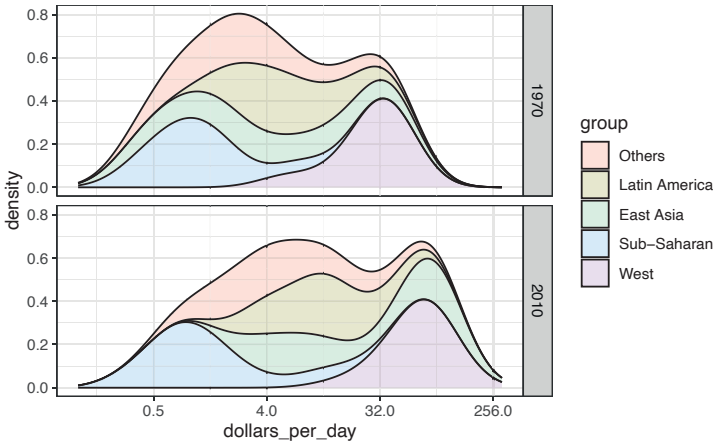
To visualize if any of the groups defined above are driving this we can quickly make a ridge plot:

```
gapminder |>
  filter(year %in% years & !is.na(dollars_per_day)) |>
  ggplot(aes(dollars_per_day, group)) +
  scale_x_continuous(trans = "log2") +
  geom_density_ridges(bandwidth = 1.5) +
  facet_grid(. ~ year)
```



Another way to achieve this is by stacking the densities on top of each other:

```
gapminder |>
    filter(year %in% years & country %in% country_list) |>
  group_by(year) |>
  mutate(weight = population/sum(population)*2) |>
  ungroup() |>
  ggplot(aes(dollars_per_day, fill = group)) +
  scale_x_continuous(trans = "log2", limits = c(0.125, 300)) +
  geom_density(alpha = 0.2, bw = 0.75, position = "stack") +
  facet_grid(year ~ .)
```



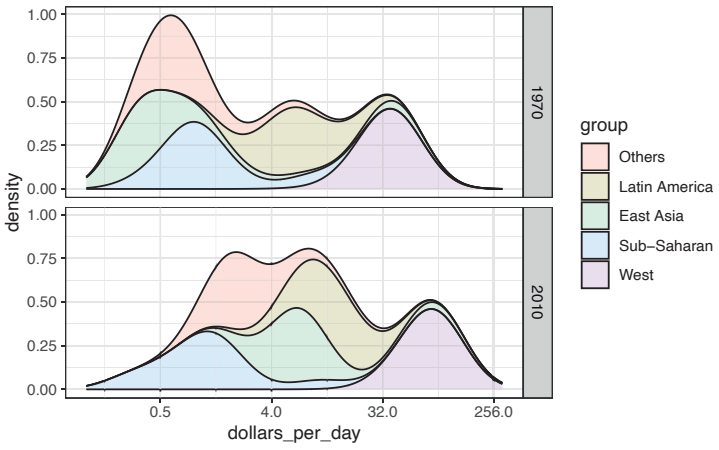
Here we can clearly see how the distributions for East Asia, Latin America, and others shift markedly to the right while Sub-Saharan Africa remains stagnant.

Notice that we order the levels of the group so that the West's density is plotted first, then Sub-Saharan Africa. Having the two extremes plotted first allows us to see the remaining bimodality better.

10.7.5 Weighted densities

As a final point, we note that these distributions weigh every country the same. So if most of the population is improving, but living in a very large country, such as China, we might not appreciate this. We can actually weight the smooth densities using the `weight` mapping argument. The plot then looks like this:

ISTUDY

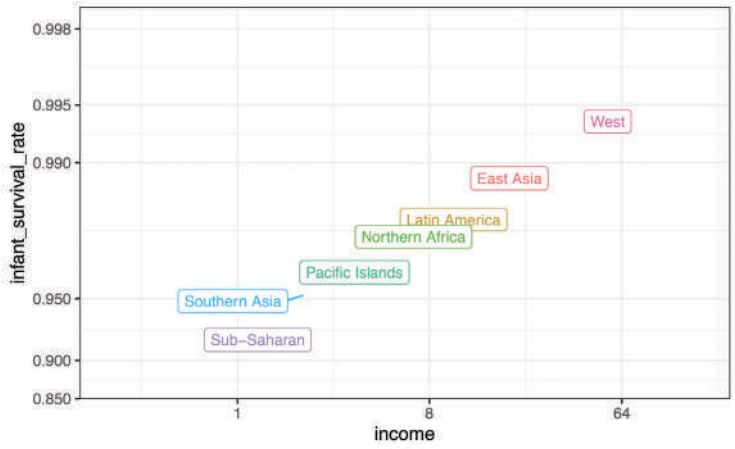


This particular figure shows very clearly how the income distribution gap is closing with most of the poor remaining in Sub-Saharan Africa.

10.8 Case study 2: the ecological fallacy

Throughout this section, we have been comparing regions of the world. We have seen that, on average, some regions do better than others. In this section, we focus on describing the importance of variability within the groups when examining the relationship between a country's infant mortality rates and average income.

We define a few more regions and compare the averages across regions:



The relationship between these two variables is almost perfectly linear with the chosen axis transformations, and the graph shows a dramatic difference. While in the West less than 0.5% of infants die, in Sub-Saharan Africa the rate is higher than 6%!

Note that the plot uses a new transformation, the logistic transformation.

ISTUDY

10.8.1 Logistic transformation

The logistic or logit transformation for a proportion or rate p is defined as:

$$f(p) = \log\left(\frac{p}{1-p}\right)$$

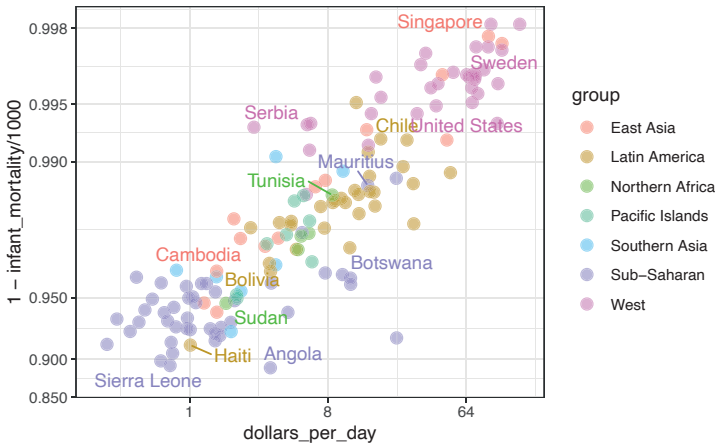
When p is a proportion or probability, the quantity that is being logged, $p/(1-p)$, is called the *odds*. In this case p is the proportion of infants that survived. The odds tell us how many more infants are expected to survive than to die. The log transformation makes this symmetric. If the rates are the same, then the log odds is 0. Fold increases or decreases turn into positive and negative increments, respectively.

This scale is useful when we want to highlight differences near 0 or 1. For survival rates this is important because a survival rate of 90% is unacceptable, while a survival of 99% is relatively good. We would much prefer a survival rate closer to 99.9%. We want our scale to highlight these difference and the logit does this. Note that 99.9/0.1 is about 10 times bigger than 99/1 which is about 10 times larger than 90/10. By using the log, these fold changes turn into constant increases.

10.8.2 Show the data

Now, back to our plot. Based on the plot above, do we conclude that a country with a low income is destined to have low survival rate? Do we conclude that survival rates in Sub-Saharan Africa are all lower than in Southern Asia, which in turn are lower than in the Pacific Islands, and so on?

Jumping to this conclusion based on a plot showing averages is referred to as the *ecological fallacy*. The almost perfect relationship between survival rates and income is only observed for the averages at the region level. Once we show all the data, we see a somewhat more complicated story:



Specifically, we see that there is a large amount of variability. We see that countries from the same regions can be quite different and that countries with the same income can have different survival rates. For example, while on average Sub-Saharan Africa had the worse health and economic outcomes, there is wide variability within that group. Mauritius and

Botswana are doing better than Angola and Sierra Leone, with Mauritius comparable to Western countries.

10.9 Case study 3: vaccines and infectious diseases

Vaccines have helped save millions of lives. In the 19th century, before herd immunization was achieved through vaccination programs, deaths from infectious diseases, such as smallpox and polio, were common. Today however, vaccination programs have become somewhat controversial despite all the scientific evidence for their importance.

The controversy started with a paper[5] published in 1988 and led by Andrew Wakefield claiming there was a link between the administration of the measles, mumps, and rubella (MMR) vaccine and the appearance of autism and bowel disease. Despite much scientific evidence contradicting this finding, sensationalist media reports and fear-mongering from conspiracy theorists led parts of the public into believing that vaccines were harmful. As a result, many parents ceased to vaccinate their children. This dangerous practice can be potentially disastrous given that the Centers for Disease Control (CDC) estimates that vaccinations will prevent more than 21 million hospitalizations and 732,000 deaths among children born in the last 20 years (see Benefits from Immunization during the Vaccines for Children Program Era — United States, 1994-2013, MMWR[6]). The 1988 paper has since been retracted and Andrew Wakefield was eventually "struck off the UK medical register, with a statement identifying deliberate falsification in the research published in The Lancet, and was thereby barred from practicing medicine in the UK." (source: Wikipedia[7]). Yet misconceptions persist, in part due to self-proclaimed activists who continue to disseminate misinformation about vaccines.

Effective communication of data is a strong antidote to misinformation and fear-mongering. In the introduction to this part of the book we showed an example, provided by a Wall Street Journal article[8], showing data related to the impact of vaccines on battling infectious diseases. Here we reconstruct that example.

10.9.1 Vaccine data

The data used for these plots were collected, organized, and distributed by the Tycho Project[9]. They include weekly reported counts for seven diseases from 1928 to 2011, from all fifty states. We include the yearly totals in the **dslabs** package:

```
library(tidyverse)
library(RColorBrewer)
library(dslabs)
names(us_contagious_diseases)
#> [1] "disease"         "state"           "year"
#> [4] "weeks_reporting" "count"           "population"
```

[5] http://www.thelancet.com/journals/lancet/article/PIIS0140-6736(97)11096-0/abstract
[6] https://www.cdc.gov/mmwr/preview/mmwrhtml/mm6316a4.htm
[7] https://en.wikipedia.org/wiki/Andrew_Wakefield
[8] http://graphics.wsj.com/infectious-diseases-and-vaccines/
[9] http://www.tycho.pitt.edu/

ISTUDY

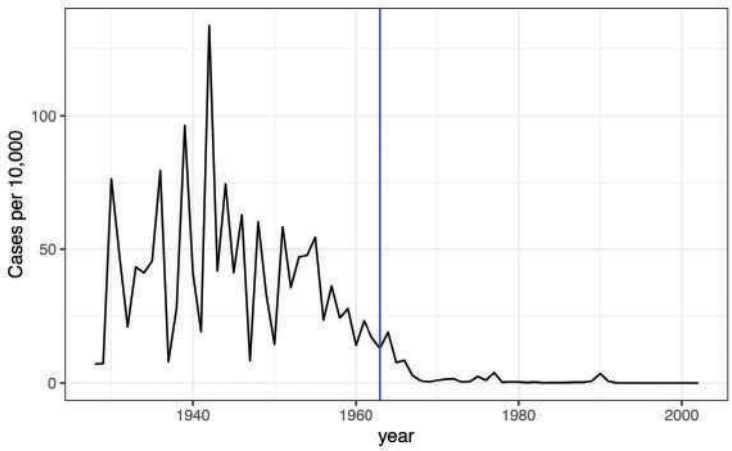
We create a temporary object `dat` that stores only the measles data, includes a per 100,000 rate, orders states by average value of disease and removes Alaska and Hawaii since they only became states in the late 1950s. Note that there is a `weeks_reporting` column that tells us for how many weeks of the year data was reported. We have to adjust for that value when computing the rate.

```
the_disease <- "Measles"
dat <- us_contagious_diseases |>
  filter(!state %in% c("Hawaii","Alaska") & disease == the_disease) |>
  mutate(rate = count / population * 10000 * 52 / weeks_reporting) |>
  mutate(state = reorder(state, ifelse(year <= 1963, rate, NA),
                         median, na.rm = TRUE))
```

10.9.2 Trend plots

We can now easily plot disease rates per year. Here are the measles data from California:

```
dat |> filter(state == "California" & !is.na(rate)) |>
  ggplot(aes(year, rate)) +
  geom_line() +
  ylab("Cases per 10,000")  +
  geom_vline(xintercept = 1963, col = "blue")
```



We add a vertical line at 1963 since this is when the vaccine was introduced [10].

10.10 Heatmaps

Now can we show data for all states in one plot? We have three variables to show: year, state, and rate. In the WSJ figure, they use the x-axis for year, the y-axis for state, and

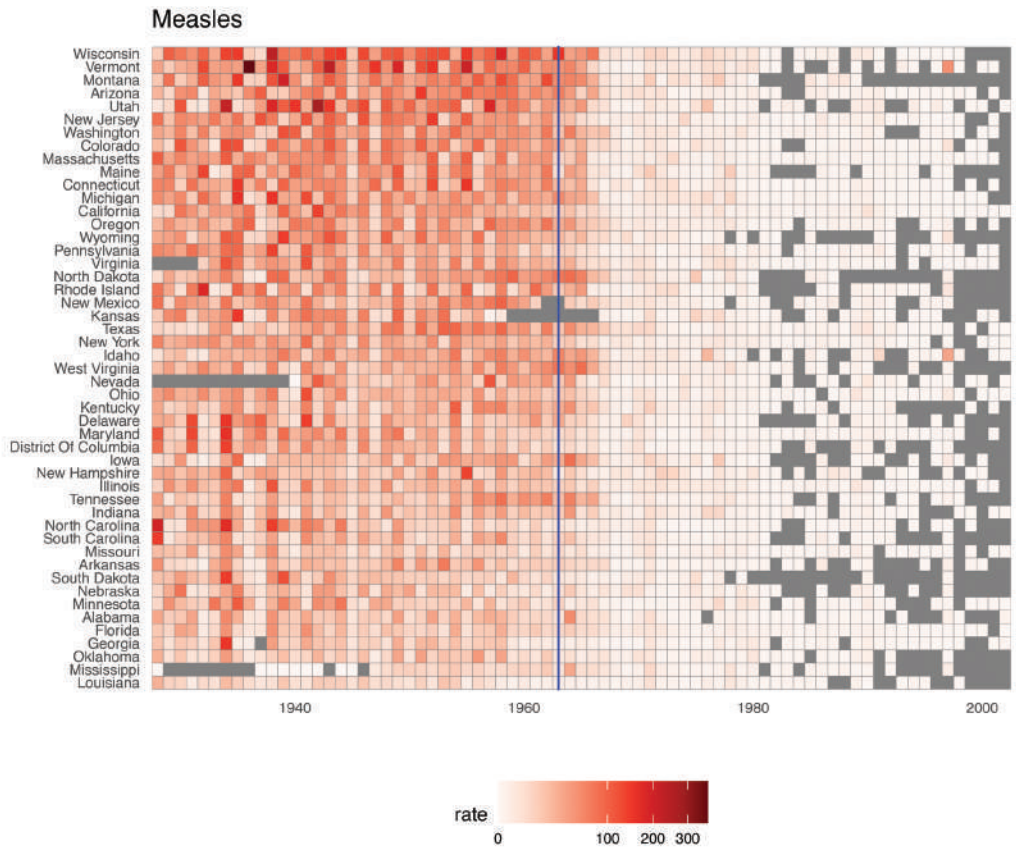
[10] Control, Centers for Disease; Prevention (2014). CDC health information for international travel 2014 (the yellow book). p. 250. ISBN 9780199948505

color hue to represent rates. However, the color scale they use, which goes from yellow to blue to green to orange to red, can be improved.

In our example, we want to use a sequential palette since there is no meaningful center, just low and high rates.

We use the geometry `geom_tile` to tile the region with colors representing disease rates. We use a square root transformation to avoid having the really high counts dominate the plot. Notice that missing values are shown in gray. Note that once a disease was pretty much eradicated, some states stopped reporting cases all together. This is why we see so much gray after 1980.

```
dat |> ggplot(aes(year, state, fill = rate)) +
  geom_tile(color = "grey50") +
  scale_x_continuous(expand = c(0,0)) +
  scale_fill_gradientn(colors = brewer.pal(9, "Reds"), trans = "sqrt") +
  geom_vline(xintercept = 1963, col = "blue") +
  theme_minimal() +
  theme(panel.grid = element_blank(),
        legend.position = "bottom",
        text = element_text(size = 8)) +
  labs(title = the_disease, x = "", y = "")
```



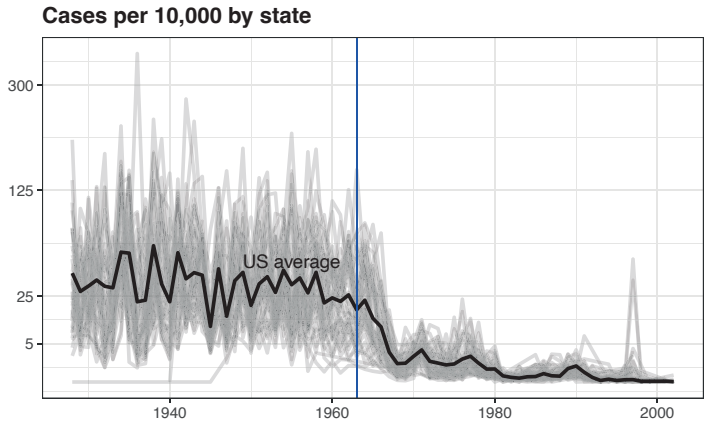
ISTUDY

This plot makes a very striking argument for the contribution of vaccines. However, one limitation of this plot is that it uses color to represent quantity, which we earlier explained makes it harder to know exactly how high values are going. Position and lengths are better cues. If we are willing to lose state information, we can make a version of the plot that shows the values with position. We can also show the average for the US, which we compute like this:

```
avg <- us_contagious_diseases |>
  filter(disease == the_disease) |> group_by(year) |>
  summarize(us_rate = sum(count, na.rm = TRUE) /
              sum(population, na.rm = TRUE) * 10000)
```

Now to make the plot we simply use the `geom_line` geometry:

```
dat |>
  filter(!is.na(rate)) |>
    ggplot() +
  geom_line(aes(year, rate, group = state),  color = "grey50",
            show.legend = FALSE, alpha = 0.2, linewidth = 1) +
  geom_line(mapping = aes(year, us_rate),  data = avg, linewidth = 1) +
  scale_y_continuous(trans = "sqrt", breaks = c(5, 25, 125, 300)) +
  ggtitle("Cases per 10,000 by state") +
  xlab("") + ylab("") +
  geom_text(data = data.frame(x = 1955, y = 50),
            mapping = aes(x, y, label = "US average"),
            color = "black") +
  geom_vline(xintercept = 1963, col = "blue")
```



In theory, we could use color to represent the categorical value state, but it is hard to pick 50 distinct colors.

10.11 Exercises

1. Reproduce the image plot we previously made but for smallpox. For this plot, do not include years in which cases were not reported in 10 or more weeks.
2. Now reproduce the time series plot we previously made, but this time following the instructions of the previous question for smallpox.
3. For the state of California, make a time series plot showing rates for all diseases. Include only years with 10 or more weeks reporting. Use a different color for each disease.
4. Now do the same for the rates for the US. Hint: compute the US rate as the total divided by total population using `summarize`.

ISTUDY



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

ISTUDY

Part III

Data Wrangling

ISTUDY



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

ISTUDY

The datasets used in this book have been made available to you as R objects, specifically as data frames. The US murders data, the reported heights data, and the Gapminder data were all data frames. These datasets come included in the **dslabs** package and we loaded them using the `data` function. Furthermore, we have made the data available in what is referred to as `tidy` form. The tidyverse packages and functions assume that the data is `tidy` and this assumption is a big part of the reason these packages work so well together.

However, very rarely in a data science project is data easily available as part of a package. We did quite a bit of work "behind the scenes" to get the original raw data into the *tidy* tables you worked with. Much more typical is for the data to be in a file, a database, or extracted from a document, including web pages, tweets, or PDFs. In these cases, the first step is to import the data into R and, when using the **tidyverse**, tidy the data. This initial step in the data analysis process usually involves several, often complicated, steps to convert data from its raw form to the *tidy* form that greatly facilitates the rest of the analysis. We refer to this process as *data wrangling*.

Here we cover several common steps of the data wrangling process including tidying data, string processing, html parsing, working with dates and times, and text analysis. Rarely are all these wrangling steps necessary in a single analysis, but as a data analysts you will likely face them all at some point. Some of the examples we use to demonstrate data wrangling techniques are based on the work we did to convert raw data into the tidy datasets provided by the **dslabs** package and used in the book as examples.

ISTUDY

11

Reshaping data

As we have seen through the book, having data in *tidy* format is what makes the tidyverse flow. After the first step in the data analysis process, importing data, a common next step is to reshape the data into a form that facilitates the rest of the analysis. The **tidyr** package, part of **tidyverse**, includes several functions that are useful for tidying data.

We will use the fertility wide format dataset described in Section 4.1 as an example in this section.

```
library(tidyverse)
library(dslabs)
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "fertility-two-countries-example.csv")
wide_data <- read_csv(filename)
```

11.1 pivot_longer

One of the most used functions in the **tidyr** package is `pivot_longer`, which is useful for converting wide data into tidy data.

As with most tidyverse functions, the `pivot_longer` function's first argument is the data frame that will be converted. Here we want to reshape the `wide_data` dataset so that each row represents a fertility observation, which implies we need three columns to store the year, country, and the observed value. In its current form, data from different years are in different columns with the year values stored in the column names. Through the `names_to` and `values_to` argument we will tell `pivot_longer` the column names we want to assign to the columns containing the current column names and observations, respectively. The default names are `name` and `value`, which are often usable choices. In this case a better choice for these two arguments would be `year` and `fertility`. Note that nowhere in the data file does it tell us this is fertility data. Instead, we deciphered this from the file name. Through `cols`, the second argument, we specify the columns containing observed values; these are the columns that will be *pivoted*. The default is to pivot all columns so, in most cases, we have to specify the columns. In our example we want columns `1960`, `1961` up to `2015`.

The code to pivot the fertility data therefore looks like this:

```
new_tidy_data <- wide_data |>
  pivot_longer(`1960`:`2015`, names_to = "year", values_to = "fertility")
```

DOI: 10.1201/9781003220923-11

We can see that the data have been converted to tidy format with columns `year` and `fertility`

```
head(new_tidy_data)
#> # A tibble: 6 x 3
#>   country year  fertility
#>   <chr>   <chr>     <dbl>
#> 1 Germany 1960       2.41
#> 2 Germany 1961       2.44
#> 3 Germany 1962       2.47
#> 4 Germany 1963       2.49
#> 5 Germany 1964       2.49
#> # i 1 more row
```

and that each year resulted in two rows since we have two countries and this column was not pivoted. A somewhat quicker way to write this code is to specify which column will **not** include in the pivot, rather than all the columns that will be pivoted:

```
new_tidy_data <- wide_data |>
  pivot_longer(-country, names_to = "year", values_to = "fertility")
```

The `new_tidy_data` object looks like the original `tidy_data` we defined this way

```
tidy_data <- gapminder |>
  filter(country %in% c("South Korea", "Germany") & !is.na(fertility)) |>
  select(country, year, fertility)
```

with just one minor difference. Can you spot it? Look at the data type of the year column. The `pivot_longer` function assumes that column names are characters. So we need a bit more wrangling before we are ready to make a plot. We need to convert the year column to be numbers:

```
new_tidy_data <- wide_data |>
  pivot_longer(-country, names_to = "year", values_to = "fertility") |>
  mutate(year = as.integer(year))
```

Now that the data is tidy, we can use this relatively simple ggplot code:

```
new_tidy_data |>
  ggplot(aes(year, fertility, color = country)) +
  geom_point()
```

11.2 `pivot_wider`

As we will see in later examples, it is sometimes useful for data wrangling purposes to convert tidy data into wide data. We often use this as an intermediate step in tidying up data. The

ISTUDY

`pivot_wider` function is basically the inverse of `pivot_longer`. The first argument is for the data, but since we are using the pipe, we don't show it. The `names_from` argument tells `pivot_wider` which variable will be used as the column names. The `values_from` argument specifies which variable to use to fill out the cells.

```
new_wide_data <- new_tidy_data |>
  pivot_wider(names_from = year, values_from = fertility)
select(new_wide_data, country, `1960`:`1967`)
#> # A tibble: 2 x 9
#>   country     `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`
#>   <chr>        <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1 Germany       2.41   2.44   2.47   2.49   2.49   2.48   2.44   2.37
#> 2 South Korea   6.16   5.99   5.79   5.57   5.36   5.16   4.99   4.85
```

Similar to `pivot_wider`, `names_from` and `values_from` default to `name` and `value`.

11.3 Separating variables

The data wrangling shown above was simple compared to what is usually required. In our example spreadsheet files, we include an illustration that is slightly more complicated. It contains two variables: life expectancy and fertility. However, the way it is stored is not tidy and, as we will explain, not optimal.

```
path <- system.file("extdata", package = "dslabs")

filename <- "life-expectancy-and-fertility-two-countries-example.csv"
filename <-  file.path(path, filename)

raw_dat <- read_csv(filename)
select(raw_dat, 1:5)
#> # A tibble: 2 x 5
#>   country     `1960_fertility` `1960_life_expectancy` `1961_fertility`
#>   <chr>                  <dbl>                  <dbl>            <dbl>
#> 1 Germany                 2.41                   69.3             2.44
#> 2 South Korea             6.16                   53.0             5.99
#> # i 1 more variable: `1961_life_expectancy` <dbl>
```

First, note that the data is in wide format. Second, notice that this table includes values for two variables, fertility and life expectancy, with the column name encoding which column represents which variable. Encoding information in the column names is not recommended but, unfortunately, it is quite common. We will put our wrangling skills to work to extract this information and store it in a tidy fashion.

We can start the data wrangling with the `pivot_longer` function, but we should no longer use the column name `year` for the new column since it also contains the variable type. We will call it `name`, the default, for now:

ISTUDY

```
dat <- raw_dat |> pivot_longer(-country)
head(dat)
#> # A tibble: 6 x 3
#>   country name                value
#>   <chr>   <chr>               <dbl>
#> 1 Germany 1960_fertility       2.41
#> 2 Germany 1960_life_expectancy 69.3
#> 3 Germany 1961_fertility       2.44
#> 4 Germany 1961_life_expectancy 69.8
#> 5 Germany 1962_fertility       2.47
#> # i 1 more row
```

The result is not exactly what we refer to as tidy since each observation is associated with two, not one, rows. We want to have the values from the two variables, fertility and life expectancy, in two separate columns. The first challenge to achieve this is to separate the `name` column into the year and the variable type. Notice that the entries in this column separate the year from the variable name with an underscore:

```
dat$name[1:5]
#> [1] "1960_fertility"       "1960_life_expectancy" "1961_fertility"
#> [4] "1961_life_expectancy" "1962_fertility"
```

Encoding multiple variables in a column name is such a common problem that the **tidyr** package includes function to separate these columns into two or more. The `separate_wider_delim` function takes three arguments: the name of the column to be separated, the names to be used for the new columns, and the character that separates the variables. So, a first attempt at separating the variable name from the year might be:

```
dat |> separate_wider_delim(name, delim = "_",
                            names = c("year", "name"))
```

However, this line of code will give an error. This is because the life expectancy names have three strings separated by `_` and the fertility names have two. This is a common problem so the `separate_wider_delim` function has arguments `too_few` and `too_many` to handle these situations. We see in the help file that the option `too_many = merge` *will merge together any additional pieces.* The following line does what we want:

```
dat |> separate_wider_delim(name, delim = "_",
                            names = c("year", "name"),
                            too_many = "merge")
#> # A tibble: 224 x 4
#>   country year  name            value
#>   <chr>   <chr> <chr>           <dbl>
#> 1 Germany 1960  fertility        2.41
#> 2 Germany 1960  life_expectancy 69.3
#> 3 Germany 1961  fertility        2.44
#> 4 Germany 1961  life_expectancy 69.8
#> 5 Germany 1962  fertility        2.47
#> # i 219 more rows
```

ISTUDY

But we are not done yet. We need to create a column for each variable and change `year` to a number. As we learned, the `pivot_wider` function can do this:

```
dat <- dat |>
  separate_wider_delim(name, delim = "_",
                       names = c("year", "name"),
                       too_many = "merge") |>
  pivot_wider() |>
  mutate(year = as.integer(year))

dat
#> # A tibble: 112 x 4
#>   country  year fertility life_expectancy
#>   <chr>   <int>     <dbl>           <dbl>
#> 1 Germany  1960      2.41            69.3
#> 2 Germany  1961      2.44            69.8
#> 3 Germany  1962      2.47            70.0
#> 4 Germany  1963      2.49            70.1
#> 5 Germany  1964      2.49            70.7
#> # i 107 more rows
```

The data is now in tidy format with one row for each observation with three variables: year, fertility, and life expectancy.

Three related function are `separate_wider_position`, `separate_wider_regex`, and `unite`. `separate_wider_position` takes a width instead of delimiter. `separate_wider_regex`, described in Section 16.4.13, provides much more control over how we separate and what we keep. The `unite` function can be thought of as the inverse of the `separate` function: it combines two columns into one.

11.4 Reshaping with data.table

In general, everything you can do with **tidyverse** can be done with **data.table** and base R which, although perhaps harder to read, it is often more flexible, faster, and more efficient. Here we show how the **data.table** approach to `pivot_longer`, `pivot_wider`, and `separate`. We will illustrate with the previously used this example:

```
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "fertility-two-countries-example.csv")
```

11.4.1 `pivot_longer` is `melt`

If in **tidyverse** we write

```
wide_data <- read_csv(filename)
new_tidy_data <- wide_data |>
  pivot_longer(-1, names_to = "year", values_to = "fertility")
```

in **data.table** we use the `melt` function.

```
library(data.table)
dt_wide_data <- fread(filename)
dt_new_tidy_data  <- melt(dt_wide_data,
                          measure.vars = 2:ncol(dt_wide_data),
                          variable.name = "year",
                          value.name = "fertility")
```

11.4.2 `pivot_wider` is `dcast`

If in **tidyverse** we write

```
new_wide_data <- new_tidy_data |>
  pivot_wider(names_from = year, values_from = fertility)
```

in **data.table** we use the `dcast` function.

```
dt_new_wide_data <- dcast(dt_new_tidy_data, formula = ... ~ year,
                          value.var = "fertility")
```

11.4.3 Separating variables

We now illustrate with this previously used example:

```
path <- system.file("extdata", package = "dslabs")
filename <- "life-expectancy-and-fertility-two-countries-example.csv"
filename <-  file.path(path, filename)
```

In **tidyverse** we wrangled using

```
raw_dat <- read_csv(filename)
dat <- raw_dat |> pivot_longer(-country) |>
  separate_wider_delim(name, delim = "_", names = c("year", "name"),
                       too_many = "merge") |>
  pivot_wider() |>
  mutate(year = as.integer(year))
```

In **data.table** we can use the `tstrsplit` function:

```
dt_raw_dat <- fread(filename)
dat_long <- melt(dt_raw_dat,
                 measure.vars = which(names(dt_raw_dat) != "country"),
                 variable.name = "name", value.name = "value")
dat_long[, c("year", "name", "name2") :=
           tstrsplit(name, "_", fixed = TRUE, type.convert = TRUE)]
dat_long[is.na(name2), name2 := ""]
dat_long[, name := paste(name, name2, sep = "_")][, name2 := NULL]
dat_wide <- dcast(dat_long, country + year ~ name, value.var = "value")
```

ISTUDY

11.5 The janitor package

The **janitor** package includes function for some of the most common steps needed to wrangle data. These are particularly useful as these tasks that are often repetitive and time-consuming. Key features include functions for examining and cleaning column names, removing empty or duplicate rows, and converting data types. It also offers capabilities to generate frequency tables and perform cross tabulations with ease. The package is designed to work seamlessly with the **tidyverse**. Here we show four examples.

Spreadsheets often use names that are not compatible with programming. The most common problem is column names with spaces. The `clean_names()` function attempts to fix this and other common problems. By default it forces variable names to be lower case and with underscore instead of space. In this example we change the variable names of the object `dat` created in the previous section and then demonstrate how this function works:

```
library(janitor)
names(dat) <- c("Country", "Year", "Fertility",  "Life Expectancy")
clean_names(dat) |> names()
#> [1] "country"         "year"            "fertility"
#> [4] "life_expectancy"
```

Another very common challenging reality is that numeric matrices are saved in spreadsheets and include a column with characters defining the row names. To fix this we have to remove the first column, but only after assigning them as vector that we will use to define rownames after converting the data frame to a matrix. The function `column_to_rows` does these operations for us and all we have to do is specify which column contains the rownames:

```
data.frame(ids = letters[1:3], x = 1:3, y = 4:6) |>
  column_to_rownames("ids") |>
  as.matrix()
#>   x y
#> a 1 4
#> b 2 5
#> c 3 6
```

Another common challenge is that spreadsheets include the column names as a first row. To quickly fix this we can "`row_to_names`":

```
x <- read.csv(file.path(path, "murders.csv"), header = FALSE) |>
  row_to_names(1)
names(x)
#> [1] "state"      "abb"        "region"     "population" "total"
```

Our final example relates to finding duplicates. A very common error in the creation of spreadsheets is that rows are duplicated. The `get_dups` function finds and reports duplicate records. By default it considers all variables, but you can also specify which ones to use.

ISTUDY

```
x <- bind_rows(x, x[1,])
get_dupes(x)
#> No variable names specified - using all columns.
#>     state abb region population total dupe_count
#> 1 Alabama  AL  South    4779736   135          2
#> 2 Alabama  AL  South    4779736   135          2
```

11.6 Exercises

1. Run the following command to define the `co2_wide` object:

```
co2_wide <- data.frame(matrix(co2, ncol = 12, byrow = TRUE)) |>
  setNames(1:12) |>
  mutate(year = as.character(1959:1997))
```

   Use the `pivot_longer` function to wrangle this into a tidy dataset. Call the column with the CO2 measurements `co2` and call the month column `month`. Call the resulting object `co2_tidy`.

2. Plot CO2 versus month with a different curve for each year using this code:

```
co2_tidy |> ggplot(aes(month, co2, color = year)) + geom_line()
```

   If the expected plot is not made, it is probably because `co2_tidy$month` is not numeric:

```
class(co2_tidy$month)
```

   Rewrite your code to make sure the month column is numeric. Then make the plot.

3. What do we learn from this plot?

   (a) CO2 measures increase monotonically from 1959 to 1997.
   (b) CO2 measures are higher in the summer and the yearly average increased from 1959 to 1997.
   (c) CO2 measures appear constant and random variability explains the differences.
   (d) CO2 measures do not have a seasonal trend.

4. Now load the `admissions` data set, which contains admission information for men and women across six majors and keep only the admitted percentage column:

```
load(admissions)
dat <- admissions |> select(-applicants)
```

ISTUDY

If we think of an observation as a major, and that each observation has two variables (men admitted percentage and women admitted percentage) then this is not tidy. Use the `pivot_wider` function to wrangle into tidy shape: one row for each major.

5. Now we will try a more advanced wrangling challenge. We want to wrangle the admissions data so that for each major we have 4 observations: `admitted_men`, `admitted_women`, `applicants_men` and `applicants_women`. The *trick* we perform here is actually quite common: first use `pivot_longer` to generate an intermediate data frame and then `pivot_wider` to obtain the tidy data we want. We will go step by step in this and the next two exercises.

   Use the `pivot_longer` function to create a `tmp` data frame with a column containing the type of observation: `admitted` or `applicants`. Call the new columns `name` and `value`.

6. Now you have an object `tmp` with columns `major`, `gender`, `name` and `value`. Note that if you combine the `name` and `gender`, we get the column names we want: `admitted_men`, `admitted_women`, `applicants_men` and `applicants_women`. Use the function `unite` to create a new column called `column_name`.

7. Now use the `pivot_wider` function to generate the tidy data with four variables for each major.

8. Now use the pipe to write a line of code that turns `admissions` to the table produced in the previous exercise.

ISTUDY

12

Joining tables

The information we need for a given analysis may not be just in one table. Here we use a simple examples to illustrate the general challenge of combining tables.

Suppose we want to explore the relationship between population size for US states and electoral votes. We have the population size in this table:

```
library(tidyverse)
library(dslabs)
head(murders)
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

and electoral votes in this one:

```
head(results_us_election_2016)
#>          state electoral_votes clinton trump others
#> 1   California              55    61.7  31.6    6.7
#> 2        Texas              38    43.2  52.2    4.5
#> 3      Florida              29    47.8  49.0    3.2
#> 4     New York              29    59.0  36.5    4.5
#> 5     Illinois              20    55.8  38.8    5.4
#> 6 Pennsylvania              20    47.9  48.6    3.6
```

Just concatenating these two tables together will not work since the order of the states is not the same.

```
identical(results_us_election_2016$state, murders$state)
#> [1] FALSE
```

The *join* functions, described below, are designed to handle this challenge.

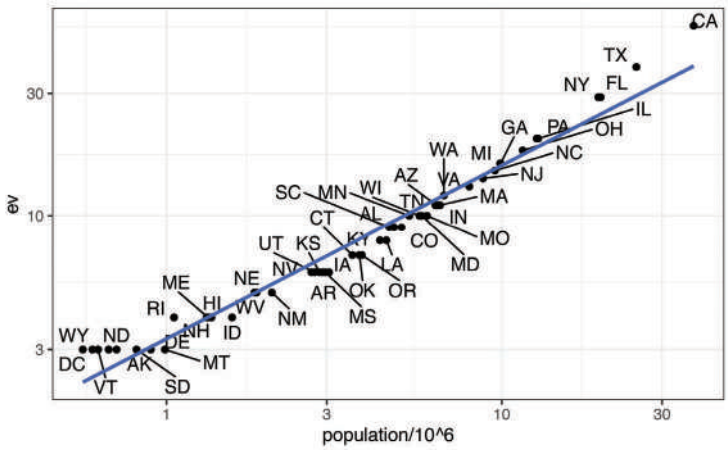
DOI: 10.1201/9781003220923-12

12.1 Joins

The *join* functions in the **dplyr** package make sure that the tables are combined so that matching rows are together. If you know SQL, you will see that the approach and syntax is very similar. The general idea is that one needs to identify one or more columns that will serve to match the two tables. Then a new table with the combined information is returned. Notice what happens if we join the two tables above by state using `left_join` (we will remove the `others` column and rename `electoral_votes` so that the tables fit on the page):

```
tab <- left_join(murders, results_us_election_2016, by = "state") |>
  select(-others) |> rename(ev = electoral_votes)
head(tab)
#>        state abb region population total ev clinton trump
#> 1    Alabama  AL  South    4779736   135  9    34.4  62.1
#> 2     Alaska  AK   West     710231    19  3    36.6  51.3
#> 3    Arizona  AZ   West    6392017   232 11    45.1  48.7
#> 4   Arkansas  AR  South    2915918    93  6    33.7  60.6
#> 5 California  CA   West   37253956  1257 55    61.7  31.6
#> 6   Colorado  CO   West    5029196    65  9    48.2  43.3
```

The data has been successfully joined and we can now, for example, make a plot to explore the relationship:



We see the relationship is close to linear with about 2 electoral votes for every million persons, but with very small states getting higher ratios.

In practice, it is not always the case that each row in one table has a matching row in the other. For this reason, we have several versions of join. To illustrate this challenge, we will take subsets of the tables above. We create the tables `tab1` and `tab2` so that they have some states in common but not all:

ISTUDY

```
tab_1 <- slice(murders, 1:6) |> select(state, population)
tab_2 <- results_us_election_2016 |>
  filter(state %in% c("Alabama", "Alaska", "Arizona",
                      "California", "Connecticut", "Delaware")) |>
  select(state, electoral_votes) |> rename(ev = electoral_votes)
```

We will use these two tables as examples in the next sections.

12.1.1 Left join

Suppose we want a table like `tab_1`, but adding electoral votes to whatever states we have available. For this, we use `left_join` with `tab_1` as the first argument. We specify which column to use to match with the `by` argument.

```
left_join(tab_1, tab_2, by = "state")
#>        state population ev
#> 1    Alabama    4779736  9
#> 2     Alaska     710231  3
#> 3    Arizona    6392017 11
#> 4   Arkansas    2915918 NA
#> 5 California   37253956 55
#> 6   Colorado    5029196 NA
```

Note that `NA`s are added to the two states not appearing in `tab_2`. Also, notice that this function, as well as all the other joins, can receive the first arguments through the pipe:

```
tab_1 |> left_join(tab_2, by = "state")
```

12.1.2 Right join

If instead of a table with the same rows as first table, we want one with the same rows as second table, we can use `right_join`:

```
tab_1 |> right_join(tab_2, by = "state")
#>         state population ev
#> 1     Alabama    4779736  9
#> 2      Alaska     710231  3
#> 3     Arizona    6392017 11
#> 4  California   37253956 55
#> 5 Connecticut         NA  7
#> 6    Delaware         NA  3
```

Now the NAs are in the column coming from `tab_1`.

12.1.3 Inner join

If we want to keep only the rows that have information in both tables, we use `inner_join`. You can think of this as an intersection:

ISTUDY

```
inner_join(tab_1, tab_2, by = "state")
#>        state population ev
#> 1    Alabama    4779736  9
#> 2     Alaska     710231  3
#> 3    Arizona    6392017 11
#> 4 California   37253956 55
```

12.1.4 Full join

If we want to keep all the rows and fill the missing parts with NAs, we can use `full_join`. You can think of this as a union:

```
full_join(tab_1, tab_2, by = "state")
#>         state population ev
#> 1     Alabama    4779736  9
#> 2      Alaska     710231  3
#> 3     Arizona    6392017 11
#> 4    Arkansas    2915918 NA
#> 5  California   37253956 55
#> 6    Colorado    5029196 NA
#> 7 Connecticut         NA  7
#> 8    Delaware         NA  3
```

12.1.5 Semi join

The `semi_join` function lets us keep the part of first table for which we have information in the second. It does not add the columns of the second:

```
semi_join(tab_1, tab_2, by = "state")
#>        state population
#> 1    Alabama    4779736
#> 2     Alaska     710231
#> 3    Arizona    6392017
#> 4 California   37253956
```

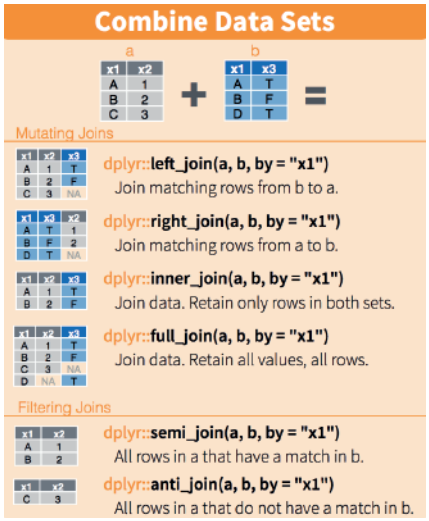
12.1.6 Anti join

The function `anti_join` is the opposite of `semi_join`. It keeps the elements of the first table for which there is no information in the second:

```
anti_join(tab_1, tab_2, by = "state")
#>      state population
#> 1 Arkansas    2915918
#> 2 Colorado    5029196
```

The following diagram summarizes the above joins:

ISTUDY



(Image courtesy of RStudio[1]. CC-BY-4.0 license[2]. Cropped from original.)

12.2 Binding

Although we have yet to use it in this book, another common way in which datasets are combined is by *binding* them. Unlike the join function, the binding functions do not try to match by a variable, but instead simply combine datasets. If the datasets don't match by the appropriate dimensions, one obtains an error.

12.2.1 Binding columns

The **dplyr** function *bind_cols* binds two objects by making them columns in a tibble. For example, we quickly want to make a data frame consisting of numbers we can use

```
bind_cols(a = 1:3, b = 4:6)
#> # A tibble: 3 x 2
#>       a     b
#>   <int> <int>
#> 1     1     4
#> 2     2     5
#> 3     3     6
```

This function requires that we assign names to the columns. Here we chose `a` and `b`.

Note that there is an R-base function `cbind` with the exact same functionality. An important difference is that `cbind` can create different types of objects, while `bind_cols` always produces a data frame.

[1] https://github.com/rstudio/cheatsheets
[2] https://github.com/rstudio/cheatsheets/blob/master/LICENSE

ISTUDY

`bind_cols` can also bind two different data frames. For example, here we break up the `tab` data frame and then bind them back together:

```
tab_1 <- tab[, 1:3]
tab_2 <- tab[, 4:6]
tab_3 <- tab[, 7:8]
new_tab <- bind_cols(tab_1, tab_2, tab_3)
head(new_tab)
#>        state abb region population total ev clinton trump
#> 1    Alabama  AL  South    4779736   135  9    34.4  62.1
#> 2     Alaska  AK   West     710231    19  3    36.6  51.3
#> 3    Arizona  AZ   West    6392017   232 11    45.1  48.7
#> 4   Arkansas  AR  South    2915918    93  6    33.7  60.6
#> 5 California  CA   West   37253956  1257 55    61.7  31.6
#> 6   Colorado  CO   West    5029196    65  9    48.2  43.3
```

12.2.2 Binding by rows

The `bind_rows` function is similar to `bind_cols`, but binds rows instead of columns:

```
tab_1 <- tab[1:2,]
tab_2 <- tab[3:4,]
bind_rows(tab_1, tab_2)
#>      state abb region population total ev clinton trump
#> 1  Alabama  AL  South    4779736   135  9    34.4  62.1
#> 2   Alaska  AK   West     710231    19  3    36.6  51.3
#> 3  Arizona  AZ   West    6392017   232 11    45.1  48.7
#> 4 Arkansas  AR  South    2915918    93  6    33.7  60.6
```

This is based on an R-base function `rbind`.

12.3 Set operators

Another set of commands useful for combining datasets are the set operators. When applied to vectors, these behave as their names suggest. Examples are `intersect`, `union`, `setdiff`, and `setequal`. However, if the **tidyverse**, or more specifically **dplyr**, is loaded, these functions can be used on data frames as opposed to just on vectors.

12.3.1 Intersect

You can take intersections of vectors of any type, such as numeric:

```
intersect(1:10, 6:15)
#> [1]  6  7  8  9 10
```

ISTUDY

or characters:

```
intersect(c("a","b","c"), c("b","c","d"))
#> [1] "b" "c"
```

The **dplyr** package includes an `intersect` function that can be applied to tables with the same column names. This function returns the rows in common between two tables. To make sure we use the **dplyr** version of `intersect` rather than the base R version, we can use `dplyr::intersect` like this:

```
tab_1 <- tab[1:5,]
tab_2 <- tab[3:7,]
dplyr::intersect(tab_1, tab_2)
#>        state abb region population total ev clinton trump
#> 1    Arizona  AZ   West    6392017   232 11    45.1  48.7
#> 2   Arkansas  AR  South    2915918    93  6    33.7  60.6
#> 3 California  CA   West   37253956  1257 55    61.7  31.6
```

12.3.2 Union

Similarly *union* takes the union of vectors. For example:

```
union(1:10, 6:15)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
union(c("a","b","c"), c("b","c","d"))
#> [1] "a" "b" "c" "d"
```

The **dplyr** package includes a version of `union` that combines all the rows of two tables with the same column names.

```
tab_1 <- tab[1:5,]
tab_2 <- tab[3:7,]
dplyr::union(tab_1, tab_2)
#>         state abb    region population total ev clinton trump
#> 1     Alabama  AL     South    4779736   135  9    34.4  62.1
#> 2      Alaska  AK      West     710231    19  3    36.6  51.3
#> 3     Arizona  AZ      West    6392017   232 11    45.1  48.7
#> 4    Arkansas  AR     South    2915918    93  6    33.7  60.6
#> 5  California  CA      West   37253956  1257 55    61.7  31.6
#> 6    Colorado  CO      West    5029196    65  9    48.2  43.3
#> 7 Connecticut  CT Northeast    3574097    97  7    54.6  40.9
```

12.3.3 `setdiff`

The set difference between a first and second argument can be obtained with `setdiff`. Unlike `intersect` and `union`, this function is not symmetric:

```
setdiff(1:10, 6:15)
#> [1] 1 2 3 4 5
```

ISTUDY

```
setdiff(6:15, 1:10)
#> [1] 11 12 13 14 15
```

As with the functions shown above, **dplyr** has a version for data frames:

```
tab_1 <- tab[1:5,]
tab_2 <- tab[3:7,]
dplyr::setdiff(tab_1, tab_2)
#>     state abb region population total ev clinton trump
#> 1 Alabama  AL  South    4779736   135  9    34.4  62.1
#> 2  Alaska  AK   West     710231    19  3    36.6  51.3
```

12.3.4 `setequal`

Finally, the function `setequal` tells us if two sets are the same, regardless of order. So notice that:

```
setequal(1:5, 1:6)
#> [1] FALSE
```

but:

```
setequal(1:5, 5:1)
#> [1] TRUE
```

The **dplyr** version checks whether data frames are equal, regardless of order of rows *or* columns:

```
dplyr::setequal(tab_1, tab_2)
#> [1] FALSE
```

12.4 Joining with data.table

The **data.table** package includes `merge`, a very efficient function for joining tables.

In **tidyverse** we joined two tables with `left_join`:

```
tab <- left_join(murders, results_us_election_2016, by = "state")
```

In **data.table** the `merge` functions works similarly:

```
library(data.table)
tab <- merge(murders, results_us_election_2016, by = "state", all.x = TRUE)
```

Instead of defining different functions for the different type of joins, `merge` uses the the logical arguments `all` (full join), `all.x` (left join), and `all.y` (right join).

ISTUDY

12.5 Exercises

1. Install and load the **Lahman** library. This database includes data related to baseball teams. It includes summary statistics about how the players performed on offense and defense for several years. It also includes personal information about the players.

   The `Batting` data frame contains the offensive statistics for all players for many years. You can see, for example, the top 10 hitters by running this code:

   ```
   library(Lahman)

   top <- Batting |>
     filter(yearID == 2016) |>
     arrange(desc(HR)) |>
     slice(1:10)

   top |> as_tibble()
   ```

   But who are these players? We see an ID, but not the names. The player names are in this table

   ```
   People |> as_tibble()
   ```

   We can see column names `nameFirst` and `nameLast`. Use the `left_join` function to create a table of the top home run hitters. The table should have `playerID`, first name, last name, and number of home runs (HR). Rewrite the object `top` with this new table.

2. Now use the `Salaries` data frame to add each player's salary to the table you created in exercise 1. Note that salaries are different every year so make sure to filter for the year 2016, then use `right_join`. This time show first name, last name, team, HR, and salary.

3. In a previous exercise, we created a tidy version of the `co2` dataset:

   ```
   co2_wide <- data.frame(matrix(co2, ncol = 12, byrow = TRUE)) |>
     setNames(1:12) |>
     mutate(year = 1959:1997) |>
     pivot_longer(-year, names_to = "month", values_to = "co2") |>
     mutate(month = as.numeric(month))
   ```

   We want to see if the monthly trend is changing, so we are going to remove the year effects and then plot the results. We will first compute the year averages. Use the `group_by` and `summarize` to compute the average co2 for each year. Save in an object called `yearly_avg`.

4. Now use the `left_join` function to add the yearly average to the `co2_wide` dataset. Then compute the residuals: observed co2 measure - yearly average.

5. Make a plot of the seasonal trends by year but only after removing the year effect.

ISTUDY

13

Parsing dates and times

We have described three main types of vectors: numeric, character, and logical. When analyzing data, we often encounter variables that are dates. Although we can represent a date with a string, for example `November 2, 2017`, once we pick a reference day, referred to as the *epoch* by computer programmers, they can be converted to numbers by calculating the number of days since the epoch. In R and Unix, the epoch is defined as January 1, 1970. So, for example, January 2, 1970 is day 1, December 31, 1969 is day −1, and November 2, 2017, is day 17,204.

Now how should we represent dates and times when analyzing data in R? We could just use days since the epoch, but then it is almost impossible to interpret. If I tell you it's November 2, 2017, you know what this means immediately. If I tell you it's day 17,204, you will be quite confused. Similar problems arise with times and even more complications can appear due to time zones. For this reason, R defines a data type just for dates and times.

13.1 The date data type

We can see an example of the data type R uses for data here:

```
library(tidyverse)
library(dslabs)
polls_us_election_2016$startdate |> head()
#> [1] "2016-11-03" "2016-11-01" "2016-11-02" "2016-11-04" "2016-11-03"
#> [6] "2016-11-03"
```

The dates look like strings, but they are not:

```
class(polls_us_election_2016$startdate)
#> [1] "Date"
```

Look at what happens when we convert them to numbers:

```
as.numeric(polls_us_election_2016$startdate) |> head()
#> [1] 17108 17106 17107 17109 17108 17108
```

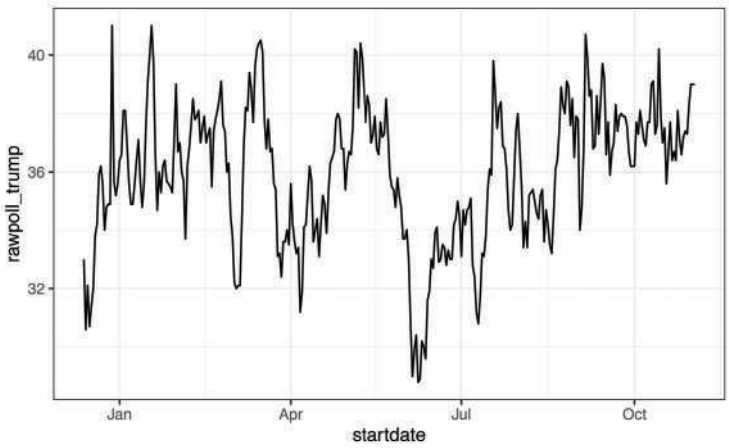
It turns them into days since the epoch. The `as.Date` function can convert a character into a date. So to see that the epoch is day 0 we can type

DOI: 10.1201/9781003220923-13

```
as.Date("1970-01-01") |> as.numeric()
#> [1] 0
```

Plotting functions, such as those in ggplot, are aware of the date format. This means that, for example, a scatterplot can use the numeric representation to decide on the position of the point, but include the string in the labels:

```
polls_us_election_2016 |> filter(pollster == "Ipsos" & state == "U.S.") |>
  ggplot(aes(startdate, rawpoll_trump)) +
  geom_line()
```



Note in particular that the month names are displayed, a very convenient feature.

13.2 The lubridate package

The **lubridate** package provides tools to work with date and times.

```
library(lubridate)
```

We will take a random sample of dates to show some of the useful things one can do:

```
set.seed(2002)
dates <- sample(polls_us_election_2016$startdate, 10) |> sort()
dates
#>  [1] "2016-05-31" "2016-08-08" "2016-08-19" "2016-09-22" "2016-09-27"
#>  [6] "2016-10-12" "2016-10-24" "2016-10-26" "2016-10-29" "2016-10-30"
```

The functions `year`, `month` and `day` extract those values:

ISTUDY

```
tibble(date = dates, month = month(dates), day = day(dates),
       year = year(dates))
#> # A tibble: 10 x 4
#>   date       month   day  year
#>   <date>     <dbl> <int> <dbl>
#> 1 2016-05-31     5    31  2016
#> 2 2016-08-08     8     8  2016
#> 3 2016-08-19     8    19  2016
#> 4 2016-09-22     9    22  2016
#> 5 2016-09-27     9    27  2016
#> # i 5 more rows
```

We can also extract the month labels:

```
month(dates, label = TRUE)
```

```
#>  [1] May Aug Aug Sep Sep Oct Oct Oct Oct Oct
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < ... < Dec
```

Another useful set of functions are the *parsers* that convert strings into dates. The function `ymd` assumes the dates are in the format YYYY-MM-DD and tries to parse as well as possible.

```
x <- c(20090101, "2009-01-02", "2009 01 03", "2009-1-4",
       "2009-1, 5", "Created on 2009 1 6", "200901 !!! 07")
ymd(x)
#> [1] "2009-01-01" "2009-01-02" "2009-01-03" "2009-01-04" "2009-01-05"
#> [6] "2009-01-06" "2009-01-07"
```

A further complication comes from the fact that dates often come in different formats in which the order of year, month, and day are different. The preferred format is to show year (with all four digits), month (two digits), and then day, or what is called the ISO 8601. Specifically we use YYYY-MM-DD so that if we order the string, it will be ordered by date. You can see the function `ymd` returns them in this format.

But, what if you encounter dates such as "09/01/02"? This could be September 1, 2002 or January 2, 2009 or January 9, 2002. In these cases, examining the entire vector of dates will help you determine what format it is by process of elimination. Once you know, you can use the many parses provided by **lubridate**.

For example, if the string is:

```
x <- "09/01/02"
```

The `ymd` function assumes the first entry is the year, the second is the month, and the third is the day, so it converts it to:

```
ymd(x)
#> [1] "2009-01-02"
```

The `mdy` function assumes the first entry is the month, then the day, then the year:

ISTUDY

```
mdy(x)
#> [1] "2002-09-01"
```

The *lubridate* package provides a function for every possibility. Here is the other common one:

```
dmy(x)
#> [1] "2002-01-09"
```

The **lubridate** package is also useful for dealing with times. In base R, you can get the current time typing `Sys.time()`. The **lubridate** package provides a slightly more advanced function, `now`, that permits you to define the time zone:

```
now()
#> [1] "2024-02-05 10:30:48 EST"
now("GMT")
#> [1] "2024-02-05 15:30:48 GMT"
```

You can see all the available time zones with `OlsonNames()` function.

We can also extract hours, minutes, and seconds:

```
now() |> hour()
#> [1] 10
now() |> minute()
#> [1] 30
now() |> second()
#> [1] 48.5
```

The package also includes a function to parse strings into times as well as parsers for time objects that include dates:

```
x <- c("12:34:56")
hms(x)
#> [1] "12H 34M 56S"
x <- "Nov/2/2012 12:34:56"
mdy_hms(x)
#> [1] "2012-11-02 12:34:56 UTC"
```

This package has many other useful functions. We describe two of these here that we find particularly useful.

The `make_date` function can be used to quickly create a date object. It can take up to seven arguments: year, month, day, hour, minute, seconds, and time zone defaulting to the epoch values on UTC time. To create an date object representing, for example, July 6, 2019 we write:

```
make_date(2019, 7, 6)
#> [1] "2019-07-06"
```

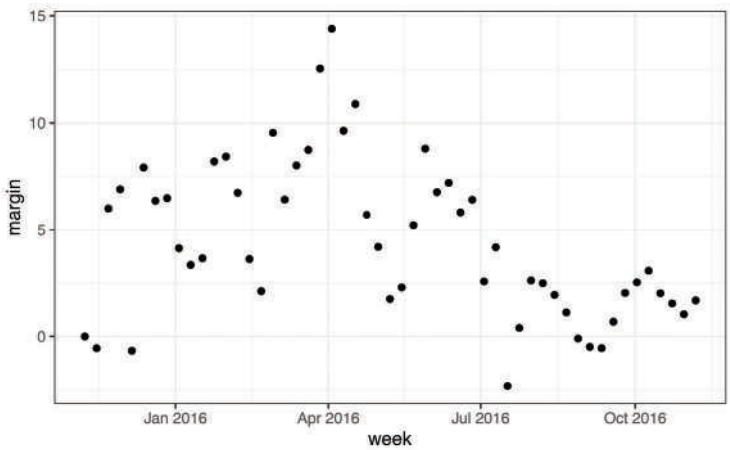
To make a vector of January 1 for the 80s we write:

ISTUDY

```
make_date(1980:1989)
#>  [1] "1980-01-01" "1981-01-01" "1982-01-01" "1983-01-01" "1984-01-01"
#>  [6] "1985-01-01" "1986-01-01" "1987-01-01" "1988-01-01" "1989-01-01"
```

Another very useful function is `round_date`. It can be used to *round* dates to nearest year, quarter, month, week, day, hour, minutes, or seconds. So if we want to group all the polls by week of the year we can do the following:

```
polls_us_election_2016 |>
  mutate(week = round_date(startdate, "week")) |>
  group_by(week) |>
  summarize(margin = mean(rawpoll_clinton - rawpoll_trump)) |>
  ggplot(aes(week, margin)) +
  geom_point()
```



Finally, you should be aware the there are useful function for computing operations on time such a `difftime`, `time_length`, and `interval`.

13.3 Dates and times with data.table

The **data.table** package includes some of the same functionality as **lubridate**. For example, it includes the `month`, and `year` functions that are the same as those in **lubridate**. The equilvalent of **lubridate**'s `day` is `mday`:

```
library(data.table)
st <- as.Date("2024-03-04")
day(st)
#> [1] 4
mday(st)
#> [1] 4
```

ISTUDY

Other similar functions included in **data.table** are `second`, `minute`, `hour`, `yday`, `wday`, `week`, `isoweek` and `quarter`.

The package also includes the classes `IDate` and `ITime`, which store dates and times more efficiently that `lubridate` and base R. This convenient for large files with date stamps. You can convert dates in the usual R format: using `as.IDate` and `as.ITime`. You can see this by using the `object.size` function:

```
object.size(polls_us_election_2016$startdate)
#> 33936 bytes
object.size(as.IDate(polls_us_election_2016$startdate))
#> 17168 bytes
```

13.4 Exercises

For these exercises we will use the following dataset:

```
library(dslabs)
head(pr_death_counts)
```

1. We want to make a plot of death counts versus date. Confirm that the `date` variable are in fact dates and not strings.
2. Plot deaths versus date.
3. What time period is represented in these data?
4. Note that after May 31, 2018, the deaths are all 0. The data is probably not entered yet. We also see a drop off starting around May 1. Redefine `dat` to exclude observations taken on or after May 1, 2018. Then, remake the plot.
5. Repeat the plot but use the day of the year on the x-axis instead of date.
6. Compute the deaths per day by month.
7. Show the deaths per days for July and for September. What do you notice?
8. Compute deaths per week and make a plot.

ISTUDY

14

Locales

Computer settings change depending on language and location, and being unaware of this possibility can make certain data-wrangling challenges difficult to overcome.

The purpose of *locales* is to group together common settings that can affect:

1. Month and day names, which are necessary for interpreting dates.
2. The standard date format, also necessary for interpreting dates.
3. The default time zone, essential for interpreting date-times.
4. Character encoding, vital for reading non-ASCII characters.
5. The symbols for decimals and number groupings, important for interpreting numerical values.

In R, a *locale* refers to a suite of settings that dictate how the system should behave with respect to cultural conventions. These settings affect the way data is formatted and presented, encompassing details such as date formatting, currency symbols, decimal separators, and other related aspects.

Locales in R affect several areas, including how character vectors are sorted, and date, number, and currency formatting. Additionally, errors, warnings, and other messages might be translated into languages other than English based on the locale.

14.1 Locales in R

To access the current locale settings in R, you can use the `Sys.getlocale()` function:

```
Sys.getlocale()
#> [1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
```

To set a specific locale, use the `Sys.setlocale()` function. For example, to set the locale to US English:

```
Sys.setlocale("LC_ALL", "en_US.UTF-8")
#> [1] "en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8"
```

The exact string to use for setting the locale (like “en_US.UTF-8”) can depend on your operating system and its configuration.

The `LC_ALL` used in the above code refers to all locale categories. R, like many systems, breaks down the locale into categories, each responsible for different aspects listed below.

DOI: 10.1201/9781003220923-14

- `LC_COLLATE`: for string collation
- `LC_TIME`: date and time formatting
- `LC_MONETARY`: currency formatting.
- `LC_MESSAGES`: system message translations.
- `LC_NUMERIC`: number formatting.

You can set the locale for each category individually if you don't want to change everything with `LC_ALL`.

> **Warning**
>
> We have shown tools to control locales. These settings are important because they affect how your data looks and behaves. However, not all of these settings are available on every computer; their availability depends on what kind of computer you have and how it's set up.
> Changing these settings, especially `LC_NUMERIC`, can lead to unexpected problems when you're working with numbers in R. For example, if you're used to using a period as a decimal point, but your locale uses a comma, this disparity can create issues when importing data.
> It is important to remember that these locale settings only last as long as one R session. If you change them while you're working, they will revert to the default settings when you close R and open it again.

14.2 The `locale` function

The **readr** package includes a `locale()` function that can be used to learn or change the current locale from within R:

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

You can see all the locales available on your system by typing:

```
system("locale -a")
```

ISTUDY

Here is what you obtain if you change the dates locale to Spanish:

```
locale(date_names = "es")
#> <locale>
#> Numbers:  123,456.78
#> Formats:  %AD / %AT
#> Timezone: UTC
#> Encoding: UTF-8
#> <date_names>
#> Days:   domingo (dom.), lunes (lun.), martes (mar.), miércoles (mié.),
#>         jueves (jue.), viernes (vie.), sábado (sáb.)
#> Months: enero (ene.), febrero (feb.), marzo (mar.), abril (abr.), mayo
#>         (may.), junio (jun.), julio (jul.), agosto (ago.),
#>         septiembre (sept.), octubre (oct.), noviembre (nov.),
#>         diciembre (dic.)
#> AM/PM:  a. m./p. m.
```

14.3 Example: wrangling a Spanish dataset

In Section 6.3.2, we noted that reading the file:

```
fn <- file.path(system.file("extdata", package = "dslabs"),
        "calificaciones.csv")
```

had a encoding different than UTF-8, the default. We used `guess_encoding` to determine the correct one:

```
guess_encoding(fn)$encoding[1]
#> [1] "ISO-8859-1"
```

and used the `locale` function to change this and read in this encoding instead:

```
dat <- read_csv(fn, locale = locale(encoding = "ISO-8859-1"))
```

This file provides homework assignment scores for seven students. The columns represent the student name, their date of birth, the time they submitted their assignment, and the score they obtained, respectively. You can see the entire file using `read_lines`:

```
read_lines(fn, locale = locale(encoding = "ISO-8859-1"))
#> [1] "\"nombre\",\"f.n.\",\"estampa\",\"puntuación\""
#> [2] "\"Beyoncé\",\"04 de septiembre de 1981\",2023-09-22 02:11:02,
#>      \"87,5\""
#> [3] "\"Blümchen\",\"20 de abril de 1980\",2023-09-22 03:23:05,
#>      \"99,0\""
#> [4] "\"João\",\"10 de junio de 1931\",2023-09-21 22:43:28,\"98,9\""
#> [5] "\"López\",\"24 de julio de 1969\",2023-09-22 01:06:59,\"88,7\""
```

ISTUDY

```
#> [6] "\"Ñengo\",\"15 de diciembre de 1981\",2023-09-21 23:35:37,
#>      \"93,1\""
#> [7] "\"Plácido\",\"24 de enero de 1941\",2023-09-21 23:17:21,\"88,7\""
#> [8] "\"Thalía\",\"26 de agosto de 1971\",2023-09-21 23:08:02,\"83,0\""
```

As an illustrative example, we will write code to compute the students age and check if they turned in their assignment by the deadline of September 21, 2023, before midnight.

We can read in the file with correct encoding like this:

```
dat <- read_csv(fn, locale = locale(encoding = "ISO-8859-1"))
```

However, notice that the last column, which is supposed to contain exam scores between 0 and 100, shows numbers larger than 800:

```
dat$puntuación
#> [1] 875 990 989 887 931 887 830
```

This happens because the scores in the file use the European decimal point, which confuses `read_csv`.

To address this issue, we can also change the encoding to use European decimals, which fixes the problem:

```
dat <- read_csv(fn, locale = locale(decimal_mark = ",",
                                    encoding = "ISO-8859-1"))
dat$puntuación
#> [1] 87.5 99.0 98.9 88.7 93.1 88.7 83.0
```

Now, to compute the student ages, let's try changing the submission times to date format:

```
library(lubridate)
#>
#> Attaching package: 'lubridate'
#> The following objects are masked from 'package:base':
#>
#>     date, intersect, setdiff, union
dmy(dat$f.n.)
#> Warning: All formats failed to parse. No formats found.
#> [1] NA NA NA NA NA NA NA
```

Nothing gets converted correctly. This is because the dates are in Spanish. We can change the locale to use Spanish as the language for dates:

```
parse_date(dat$f.n., format = "%d de %B de %Y",
locale = locale(date_names = "es"))
#> [1] "1981-09-04" "1980-04-20" "1931-06-10" "1969-07-24" "1981-12-15"
#> [6] "1941-01-24" "1971-08-26"
```

We can also reread the file using the correct locales:

ISTUDY

```
dat <- read_csv(fn, locale = locale(date_names = "es",
                                    date_format = "%d de %B de %Y",
                                    decimal_mark = ",",
                                    encoding = "ISO-8859-1"))
```

Computing the students' ages is now straightforward:

```
time_length(today() - dat$f.n., unit = "years") |> floor()
#> [1] 42 43 92 54 42 83 52
```

Finally, let's check which students turned in their homework past the deadline of September 22:

```
dat$estampa >= make_date(2023, 9, 22)
#> [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

We see that two students where late. However, with times we have to be particularly careful as some functions default to the UTC timezone:

```
tz(dat$estampa)
#> [1] "UTC"
```

If we change to the timezone to Eastern Standard Time (EST), we see no one was late:

```
with_tz(dat$estampa, tz =  "EST") >= make_date(2023, 9, 22)
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

14.4 Exercises

1. Load the **lubridate** package and set the locale to French for this exercise.
2. Create a numeric vector containing the following numbers: 12345.67, 9876.54, 3456.78, and 5432.10.
3. Use the `format()` function to format the numeric vector as currency, displaying the values in Euros. Ensure that the decimal point is represented correctly according to the French locale. Print the formatted currency values.
4. Create a date vector with three dates: July 14, 1789, January 1, 1803, and July 5, 1962. Use the `format()` function to format the date vector in the "dd Month yyyy" format, where "Month" should be displayed in the French language. Ensure that the month names are correctly translated according to the French locale. Print the formatted date values.
5. Reset the locale to the default setting (e.g., "C" or "en_US.UTF-8") to revert to the standard formatting.
6. Repeat steps 2–4 for the numeric vector, and steps 5–7 for the date vector to observe the standard formatting.

ISTUDY

15

Extracting data from the web

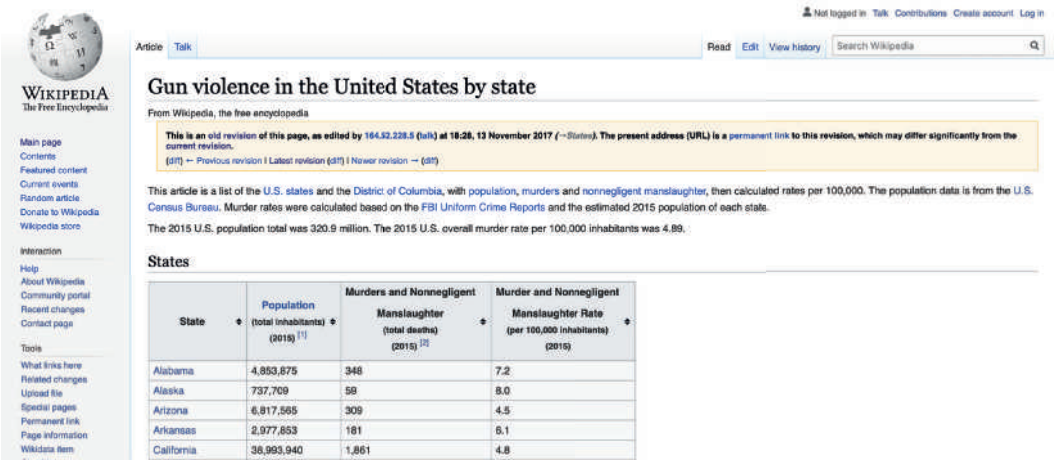
In today's digital age, the internet serves as a treasure trove of data. This chapter describes approaches for retrieving this data and preparing for data analysis. We focus on two primary methodologies: scraping html and API integration. Scraping html allows us to programmatically navigate and extract data from web pages, transforming the unstructured content of the HTML pages into structured data ready for analysis. On the other hand, APIs provide a more direct and efficient and structured access to data provided by web services. This chapter aims to introduce the basics needed to starting leveraging the web's vast data resources.

15.1 Scraping HTML

The data we need to answer a question is not always in a spreadsheet ready for us to read. For example, the US murders dataset we used in the R Basics chapter originally comes from this Wikipedia page:

```
url <- paste0("https://en.wikipedia.org/w/index.php?title=",
              "Gun_violence_in_the_United_States_by_state",
              "&direction=prev&oldid=810166167")
```

You can see the data table when you visit the webpage:



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store
Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page
Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item

Gun violence in the United States by state

From Wikipedia, the free encyclopedia

This is an old revision of this page, as edited by 164.52.228.5 (talk) at 18:28, 13 November 2017 (→States). The present address (URL) is a permanent link to this revision, which may differ significantly from the current revision.
(diff) ← Previous revision | Latest revision (diff) | Newer revision → (diff)

This article is a list of the U.S. states and the District of Columbia, with population, murders and nonnegligent manslaughter, then calculated rates per 100,000. The population data is from the U.S. Census Bureau. Murder rates were calculated based on the FBI Uniform Crime Reports and the estimated 2015 population of each state.

The 2015 U.S. population total was 320.9 million. The 2015 U.S. overall murder rate per 100,000 inhabitants was 4.89.

States

State	Population (total inhabitants) (2015) [1]	Murders and Nonnegligent Manslaughter (total deaths) (2015) [2]	Murder and Nonnegligent Manslaughter Rate (per 100,000 inhabitants) (2015)
Alabama	4,853,875	348	7.2
Alaska	737,709	59	8.0
Arizona	6,817,565	309	4.5
Arkansas	2,977,853	181	6.1
California	38,993,940	1,861	4.8

DOI: 10.1201/9781003220923-15

(Web page courtesy of Wikipedia[1]. CC-BY-SA-3.0 license[2]. Screenshot of part of the page.)

Web scraping, or *web harvesting*, is the term we use to describe the process of extracting data from a website. The reason we can do this is because the information used by a browser to render webpages is received as a text file from a server. The text is code written in hyper text markup language (HTML). Every browser has a way to show the html source code for a page, each one different. On Chrome, you can use Control-U on a PC and command+alt+U on a Mac. You will see something like this:



15.1.1 HTML

Because this code is accessible, we can download the HTML file, import it into R, and then write programs to extract the information we need from the page. However, once we look at HTML code, this might seem like a daunting task. But we will show you some convenient tools to facilitate the process. To get an idea of how it works, here are a few lines of code from the Wikipedia page that provides the US murders data:

```
<table class="wikitable sortable">
<tr>
<th>State</th>
<th><a href="/wiki/List_of_U.S._states_and_territories_by_population"
title="List of U.S. states and territories by population">Population</a>
   <br />
<small>(total inhabitants)</small><br />
<small>(2015)</small> <sup id="cite_ref-1" class="reference">
<a href="#cite_note-1">[1]</a></sup></th>
<th>Murders and Nonnegligent
<p>Manslaughter<br />
```

[1] https://en.wikipedia.org/w/index.php?title=Gun_violence_in_the_United_States_by_state&direction=prev&oldid=810166167

[2] https://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Commons_Attribution-ShareAlike_3.0_Unported_License

ISTUDY

```
<small>(total deaths)</small><br />
<small>(2015)</small> <sup id="cite_ref-2" class="reference">
<a href="#cite_note-2">[2]</a></sup></p>
</th>
<th>Murder and Nonnegligent
<p>Manslaughter Rate<br />
<small>(per 100,000 inhabitants)</small><br />
<small>(2015)</small></p>
</th>
</tr>
<tr>
<td><a href="/wiki/Alabama" title="Alabama">Alabama</a></td>
<td>4,853,875</td>
<td>348</td>
<td>7.2</td>
</tr>
<tr>
<td><a href="/wiki/Alaska" title="Alaska">Alaska</a></td>
<td>737,709</td>
<td>59</td>
<td>8.0</td>
</tr>
<tr>
```

You can actually see the data, except data values are surrounded by html code such as `<td>`. We can also see a pattern of how it is stored. If you know HTML, you can write programs that leverage knowledge of these patterns to extract what we want. We also take advantage of a language widely used to make webpages look "pretty" called Cascading Style Sheets (CSS). We say more about this in Section 15.1.3.

Although we provide tools that make it possible to scrape data without knowing HTML, it is useful to learn some HTML and CSS. Not only does this improve your scraping skills, but it might come in handy if you are creating a webpage to showcase your work. There are plenty of online courses and tutorials for learning these. Two examples are Codeacademy[3] and W3schools[4].

15.1.2 The rvest package

The **tidyverse** provides a web harvesting package called **rvest**. The first step using this package is to import the webpage into R. The package makes this quite simple:

```
library(tidyverse)
library(rvest)
h <- read_html(url)
```

Note that the entire Murders in the US Wikipedia webpage is now contained in `h`. The class of this object is:

[3] https://www.codecademy.com/learn/learn-html
[4] https://www.w3schools.com/

ISTUDY

```
class(h)
#> [1] "xml_document" "xml_node"
```

The **rvest** package is actually more general; it handles XML documents. XML is a general markup language (that's what the ML stands for) that can be used to represent any kind of data. HTML is a specific type of XML specifically developed for representing webpages. Here we focus on HTML documents.

Now, how do we extract the table from the object `h`? If you were to print `h`, we would see information about the object that is not very informative. We can see all the code that defines the downloaded webpage using the `html_text` function like this:

```
html_text(h)
```

We don't show the output here because it includes thousands of characters. But if we look at it, we can see the data we are after are stored in an HTML table: you can see this in this line of the HTML code above `<table class="wikitable sortable">`. The different parts of an HTML document, often defined with a message in between `<` and `>` are referred to as *nodes*. The **rvest** package includes functions to extract nodes of an HTML document: `html_nodes` extracts all nodes of different types and `html_node` extracts the first one. To extract the tables from the html code we use:

```
tab <- h |> html_nodes("table")
```

Now, instead of the entire webpage, we just have the html code for the tables in the page:

```
tab
#> {xml_nodeset (2)}
#> [1] <table class="wikitable sortable"><tbody>\n<tr>\n<th>State\n</th> ...
#> [2] <table class="nowraplinks hlist mw-collapsible mw-collapsed navbo ...
```

The table we are interested is the first one:

```
tab[[1]]
#> {html_node}
#> <table class="wikitable sortable">
#> [1] <tbody>\n<tr>\n<th>State\n</th>\n<th>\n<a href="/wiki/List_of_U.S ...
```

This is clearly not a tidy dataset, not even a data frame. In the code above, you can definitely see a pattern and writing code to extract just the data is very doable. In fact, **rvest** includes a function just for converting HTML tables into data frames:

```
tab <- tab[[1]] |> html_table()
class(tab)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

ISTUDY

We are now much closer to having a usable data table:

```
tab <- tab |> setNames(c("state", "population", "total", "murder_rate"))
head(tab)
#> # A tibble: 6 x 4
#>   state      population total murder_rate
#>   <chr>      <chr>      <chr>       <dbl>
#> 1 Alabama    4,853,875  348           7.2
#> 2 Alaska     737,709    59            8
#> 3 Arizona    6,817,565  309           4.5
#> 4 Arkansas   2,977,853  181           6.1
#> 5 California 38,993,940 1,861         4.8
#> # i 1 more row
```

We still have some wrangling to do. For example, we need to remove the commas and turn characters into numbers. Before continuing with this, we will learn a more general approach to extracting information from web sites.

15.1.3 CSS selectors

The default look of a webpage made with the most basic HTML is quite unattractive. The aesthetically pleasing pages we see today are made using CSS to define the look and style of webpages. The fact that all pages for a company have the same style usually results from their use of the same CSS file to define the style. The general way these CSS files work is by defining how each of the elements of a webpage will look. The title, headings, itemized lists, tables, and links, for example, each receive their own style including font, color, size, and distance from the margin. CSS does this by leveraging patterns used to define these elements, referred to as *selectors*. An example of such a pattern, which we used above, is `table`, but there are many, many more.

If we want to grab data from a webpage and we happen to know a selector that is unique to the part of the page containing this data, we can use the `html_nodes` function. However, knowing which selector can be quite complicated. In fact, the complexity of webpages has been increasing as they become more sophisticated. For some of the more advanced ones, it seems almost impossible to find the nodes that define a particular piece of data. However, selector gadgets actually make this possible.

SelectorGadget[5] is piece of software that allows you to interactively determine what CSS selector you need to extract specific components from the webpage. If you plan on scraping data other than tables from html pages, we highly recommend you install it. A Chrome extension is available which permits you to turn on the gadget and then, as you click through the page, it highlights parts and shows you the selector you need to extract these parts. There are various demos of how to do this including **rvest** author Hadley Wickham's vignette[6] and other tutorials based on the vignette[7].

[5] http://selectorgadget.com/

[6] https://rvest.tidyverse.org/articles/selectorgadget.html

[7] https://www.analyticsvidhya.com/blog/2017/03/beginners-guide-on-web-scraping-in-r-using-rvest-with-hands-on-knowledge/

ISTUDY

15.2 JSON

Sharing data on the internet has become more and more common. Unfortunately, providers use different formats, which makes it harder for data analysts to wrangle data into R. Yet there are some standards that are also becoming more common. Currently, a format that is widely being adopted is the JavaScript Object Notation or JSON. Because this format is very general, it is nothing like a spreadsheet. This JSON file looks more like the code you use to define a list. Here is an example of information stored in a JSON format:

```
#> [
#>   {
#>     "name": "Miguel",
#>     "student_id": 1,
#>     "exam_1": 85,
#>     "exam_2": 86
#>   },
#>   {
#>     "name": "Sofia",
#>     "student_id": 2,
#>     "exam_1": 94,
#>     "exam_2": 93
#>   },
#>   {
#>     "name": "Aya",
#>     "student_id": 3,
#>     "exam_1": 87,
#>     "exam_2": 88
#>   },
#>   {
#>     "name": "Cheng",
#>     "student_id": 4,
#>     "exam_1": 90,
#>     "exam_2": 91
#>   }
#> ]
```

The file above actually represents a data frame. To read it, we can use the function `fromJSON` from the **jsonlite** package. Note that JSON files are often made available via the internet. Several organizations provide a JSON API or a web service that you can connect directly to and obtain data. Here is an example providing information Nobel prize winners:

```
library(jsonlite)
nobel <- fromJSON("http://api.nobelprize.org/v1/prize.json")
```

This downloads a list. The first argument, named "prizes" is a table with information about Nobel prize winners. Each row holds corresponds to a particular year and category. The "laureates" column holds a list with a data frame for each winner with columns for id, firstname, surname, and motivation.

ISTUDY

```
nobel$prizes |>
  filter(category == "literature" & year == "1971") |>
  pull(laureates) |>
  first() |>
  select(id, firstname, surname)
#>    id firstname surname
#> 1 645     Pablo  Neruda
```

You can learn much more by examining tutorials and help files from the **jsonlite** package. This package is intended for relatively simple tasks such as converting data into tables. For more flexibility, we recommend the **rjson** package.

15.3 Data APIs

An Application Programming Interface (API) is a set of rules and protocols that allows different software entities to communicate with each other. It defines methods and data formats that software components should use when requesting and exchanging information. APIs play a crucial role in enabling the integration that make today's software so interconnected and versatile.

15.3.1 API types and concepts

There are several types of APIs. The main ones related to retrieving data are:

- **Web Services** – Often built using protocols like HTTP/HTTPS. Commonly used to enable applications to communicate with each other over the web. For instance, a weather application for a smartphone may use a web API to request weather data from a remote server.
- **Database APIs** – Enable communication between an application and a database, SQL-based calls for example.

Some key concepts associated with APIs:

- **Endpoints**: Specific functions available through the API. For web APIs, an endpoint is usually a specific URL where the API can be accessed.
- **Methods**: Actions that can be performed. In web APIs, these often correspond to HTTP methods like GET, POST, PUT, or DELETE.
- **Requests and Responses**: The act of asking the API to perform its function is a *request*. The data it returns is the *response*.
- **Rate Limits**: Restrictions on how often you can call the API, often used to prevent abuse or overloading of the service.
- **Authentication and Authorization**: Mechanisms to ensure that only approved users or applications can use the API. Common methods include *API keys*, *OAuth*, or *Jason Web Tokens* (JWT).
- **Data Formats**: Many web APIs exchange data in a specific format, often JSON or CSV.

ISTUDY

Here now describe the **httr2** package that facilitates interacionts between R and HTTP web services.

15.3.2 The httr2 package

HTTP is the most widely used protocol for data sharing through the internet. The **httr2** package provides functions to work with HTTP requests. One of the core functions in this package is `request`, which is used to form request to send to web services. The `req_perform` function sends the request.

This `request` function forms an HTTP GET request to the specified URL. Typically, HTTP GET requests are used to retrieve information from a server based on the provided URL.

The function returns an object of class `response`. This object contains all the details of the server's response, including status code, headers, and content. You can then use other **httr2** functions to extract or interpret information from this response.

Let's say you want to retrieve COVID-19 deaths by state from the CDC. By visiting their data catalog[8] you can search for datasets and find that the data is provided through this API:

```
url <- "https://data.cdc.gov/resource/muzy-jte6.csv"
```

We can then make create and perform a request like this:

```
library(httr2)
response <- request(url) |> req_perform()
```

We can see the results of the request by looking at the returned object.

```
response
#> <httr2_response>
#> GET https://data.cdc.gov/resource/muzy-jte6.csv
#> Status: 200 OK
#> Content-Type: text/csv
#> Body: In memory (210808 bytes)
```

To extract the body, which is where the data are, we can use `resp_body_string` and send the result, a comma delimited string, to `read_csv`

```
library(readr)
tab <- response |> resp_body_string() |> read_csv()
```

We note that the returned object is only 1000 entries. API often limit how much you can download. The documentation for this API[9] explains that we can change this limit through the `$limit` parameters. We can use the `req_url_path_append` to add this to our request:

[8] https://data.cdc.gov
[9] https://dev.socrata.com/docs/queries/

ISTUDY

```
response <- request(url) |>
  req_url_path_append("?$limit=100000") |>
  req_perform()
```

The CDC service returns data in csv format but a more common format used by web services is JSON. The CDC also provides data in json format through a the url:

```
url <- "https://data.cdc.gov/resource/muzy-jte6.json"
```

To extract the data table we use the `fromJSON` function from the **jsonlite** package.

```
tab <- request(url) |>
   req_perform() |>
   resp_body_string() |>
   fromJSON(flatten = TRUE)
```

When working with APIs, it's essential to check the API's documentation for rate limits, required headers, or authentication methods. The `httr2` package provides tools to handle these requirements, such as setting headers or authentication parameters.

15.4 Exercises

1. Visit the following web page: https://web.archive.org/web/20181024132313/http://www.stevetheump.com/Payrolls.htm

   Notice there are several tables. Say we are interested in comparing the payrolls of teams across the years. The next few exercises take us through the steps needed to do this.

   Start by applying what you learned to read in the website into an object called `h`.

2. Note that, although not very useful, we can actually see the content of the page by typing:

   ```
   html_text(h)
   ```

   The next step is to extract the tables. For this, we can use the `html_nodes` function. We learned that tables in html are associated with the `table` node. Use the `html_nodes` function to extract all tables. Store it in an object `nodes`.

3. The `html_nodes` function returns a list of objects of class `xml_node`. We can see the content of each one using, for example, the `html_text` function. You can see the content for an arbitrarily picked component like this:

   ```
   html_text(nodes[[8]])
   ```

   If the content of this object is an html table, we can use the `html_table` function to convert it to a data frame. Use the `html_table` function to convert the 8th entry of `nodes` into a table.

ISTUDY

4. Repeat the above for the first 4 components of `nodes`. Which of the following are payroll tables:

   (a) All of them.
   (b) 1
   (c) 2
   (d) 2–4

5. Repeat the above for the first **last** 3 components of `nodes`. Which of the following is true:

   (a) The last entry in `nodes` shows the average across all teams through time, not payroll per team.
   (b) All three are payroll per team tables.
   (c) All three are like the first entry, not a payroll table.
   (d) All of the above.

6. We have learned that the first and last entries of `nodes` are not payroll tables. Redefine `nodes` so that these two are removed.

7. We saw in the previous analysis that the first table node is not actually a table. This happens sometimes in html because tables are used to make text look a certain way, as opposed to storing numeric values. Remove the first component and then use `sapply` and `html_table` to convert each node in `nodes` into a table. Note that in this case, `sapply` will return a list of tables. You can also use `lapply` to assure that a list is applied.

8. Look through the resulting tables. Are they all the same? Could we just join them with `bind_rows`?

9. Create two tables, call them `tab_1` and `tab_2` using the 10th and 19th tables in `nodes`.

10. Use a `full_join` function to combine these two tables. Before you do this you will have to fix the missing header problem. You will also need to make the names match.

11. After joining the tables, you see several `NA`s. This is because some teams are in one table and not the other. Use the `anti_join` function to get a better idea of why this is happening.

12. We see see that one of the problems is that Yankees are listed as both *N.Y. Yankees* and *NY Yankees*. In the next section, we will learn efficient approaches to fixing problems like this. Here we can do it "by hand" as follows:

```
tab_1 <- tab_1 |>
  mutate(Team = ifelse(Team == "N.Y. Yankees", "NY Yankees", Team))
```

    Now join the tables and show only Oakland and the Yankees and the payroll columns.

13. Advanced: extract the titles of the movies that won Best Picture from IMDB[10].

[10] https://m.imdb.com/chart/bestpicture/

ISTUDY

16

String processing

One of the most common data-wrangling challenges involves extracting numeric data contained in character strings and converting them into the numeric representations required to make plots, compute summaries, or fit models in R. Also common is processing unorganized text into meaningful variable names or categorical variables. Many of the string processing challenges a data scientist faces are unique and often unexpected. It is therefore quite ambitious to write a comprehensive section on this topic. Here we use a series of case studies that help us demonstrate how string processing is a necessary step for many data-wrangling challenges. Specifically, we describe the process of converting the original, *raw* data from which we extracted the `murders`, `heights`, and `research_funding_rates` examples into the data frames we have studied in this book.

By going over these case studies, we will cover some of the most common tasks in string processing including extracting numbers from strings, removing unwanted characters from text, finding and replacing characters, extracting specific parts of strings, converting free-form text to more uniform formats, and splitting strings into multiple values.

Base R includes functions to perform many of these tasks. The **stringi** package adds significant functionality over what is available in base R, especially for complex and diverse text processing needs. The **stringr** package provides consistent, simple, and user friendly wrappers around the **stringi** package. For example, in **stringr**, all the string processing functions start with `str_`. This means that if you type `str_` and hit tab, R will auto-complete and show all the available functions. As a result, we don't necessarily have to memorize all the function names. Another advantage is that in the functions in this package the string being processed is always the first argument, which means we can more easily use the pipe. Therefore, we will start by describing how to use the functions in the **stringr** package.

Most of the examples will come from the second case study which deals with self-reported heights by students, and most of the chapter is dedicated to learning regular expressions (regex) and functions in the **stringr** package.

16.1 The stringr package

```
library(tidyverse)
library(stringr)
```

In general, string processing tasks can be divided into **detecting**, **locating**, **extracting**, or **replacing** patterns in strings. We will see several examples. The table below includes the functions available to you in the **stringr** package. We split them by task. We also include the base R equivalent when available.

DOI: 10.1201/9781003220923-16

All these functions take a character vector as first argument. Also, for each function, operations are vectorized: the operation gets applied to each string in the vector.

Finally, we mention *groups* in this table. These will be explained in Section 16.4.9.

stringr	Task	Description	Base R
`str_detect`	Detect	Is the pattern in the string?	`grepl`
`str_which`	Detect	Returns the index of entries that contain the pattern.	`grep`
`str_subset`	Detect	Returns the subset of strings that contain the pattern.	`grep` with `value = TRUE`
`str_locate`	Locate	Returns positions of first occurrence of the pattern in a string.	`regexpr`
`str_locate_all`	Locate	Returns position of all occurrences of the pattern in a string.	`gregexpr`
`str_view`	Locate	Show the first part of the string that matches the pattern.	
`str_view_all`	Locate	Show all the parts of the string that match the pattern.	
`str_extract`	Extract	Extract the first part of the string that matches the pattern.	
`str_extract_all`	Extract	Extract all parts of the string that match the pattern.	
`str_match`	Extract	Extract first part of the string that matches the pattern and the groups defined by the pattern.	
`str_match_all`	Extract	Extract all parts of the string that match the pattern and the groups defined by the pattern.	
`str_sub`	Extract	Extract a substring.	`substring`
`str_split`	Extract	Split a string into a list with parts separated by a pattern.	`strsplit`
`str_split_fixed`	Extract	Split a string into a matrix with a fixed number of parts separated by a pattern.	`strsplit` with `fixed = TRUE`
`str_count`	Describe	Count number of times a pattern appears in a string.	
`str_length`	Describe	Number of character in string.	`nchar`
`str_replace`	Replace	Replace first part of a string matching a pattern with another.	
`str_replace_all`	Replace	Replace all parts of a string matching a pattern with another.	`gsub`
`str_to_upper`	Replace	Change all characters to upper case.	`toupper`
`str_to_lower`	Replace	Change all characters to lower case.	`tolower`
`str_to_title`	Replace	Change first character of each word to upper and rest to lower case.	
`str_replace_na`	Replace	Replace all `NA`s with a new value.	
`str_trim`	Replace	Remove white space from start and end of string.	
`str_c`	Manipulate	Join multiple strings.	`paste0`
`str_conv`	Manipulate	Change the encoding of the string.	

ISTUDY

stringr	Task	Description	Base R
`str_sort`	Manipulate	Sort the vector in alphabetical order.	`sort`
`str_order`	Manipulate	Provide index needed to order the vector in alphabetical order.	`order`
`str_trunc`	Manipulate	Truncate a string to a fixed size.	
`str_pad`	Manipulate	Add white space to string to make it a fixed size.	
`str_dup`	Manipulate	Repeat a string.	`rep` then `paste`
`str_wrap`	Manipulate	Wrap things into formatted paragraphs.	
`str_interp`	Manipulate	String interpolation.	`sprintf`

16.2 Case study 1: self-reported heights

The **dslabs** package includes the raw data from which the heights dataset was obtained. You can load it like this:

```
library(dslabs)
head(reported_heights)
#>            time_stamp    sex height
#> 1 2014-09-02 13:40:36   Male     75
#> 2 2014-09-02 13:46:59   Male     70
#> 3 2014-09-02 13:59:20   Male     68
#> 4 2014-09-02 14:51:53   Male     74
#> 5 2014-09-02 15:16:15   Male     61
#> 6 2014-09-02 15:16:16 Female     65
```

These heights were obtained using a web form in which students were asked to enter their heights. They could enter anything, but the instructions asked for *height in inches*, a number. We compiled 1,095 submissions, but unfortunately the column vector with the reported heights had several non-numeric entries and as a result became a character vector:

```
class(reported_heights$height)
#> [1] "character"
```

If we try to parse it into numbers, we get a warning:

```
x <- as.numeric(reported_heights$height)
#> Warning: NAs introduced by coercion
```

ISTUDY

Although most values appear to be height in inches as requested we do end up with many `NA`s:

```
sum(is.na(x))
#> [1] 81
```

Here are some of the entries that are not successfully converted:

```
reported_heights |>
  mutate(new_height = as.numeric(height)) |>
  filter(is.na(new_height)) |>
  head(n = 10)
#>              time_stamp    sex                 height new_height
#> 1  2014-09-02 15:16:28   Male                  5' 4"         NA
#> 2  2014-09-02 15:16:37 Female                  165cm         NA
#> 3  2014-09-02 15:16:52   Male                    5'7         NA
#> 4  2014-09-02 15:16:56   Male                  >9000         NA
#> 5  2014-09-02 15:16:56   Male                   5'7"         NA
#> 6  2014-09-02 15:17:09 Female                   5'3"         NA
#> 7  2014-09-02 15:18:00   Male 5 feet and 8.11 inches         NA
#> 8  2014-09-02 15:19:48   Male                   5'11         NA
#> 9  2014-09-04 00:46:45   Male                  5'9''         NA
#> 10 2014-09-04 10:29:44   Male                 5'10''         NA
```

We immediately see what is happening. Some of the students did not report their heights in inches as requested. We could discard these data and continue. However, many of the entries follow patterns that, in principle, we can easily convert to inches. For example, in the output above, we see various cases that use the format `x'y"` or `x'y''` with `x` and `y` representing feet and inches, respectively. Each one of these cases can be read and converted to inches by a human, for example `5'4"` is `5*12 + 4 = 64`. So we could fix all the problematic entries *by hand*. However, humans are prone to making mistakes, so an automated approach is preferable. Also, because we plan on continuing to collect data, it will be convenient to write code that automatically corrects entries entered in error.

A first step in this type of task is to survey the problematic entries and try to define specific patterns followed by a large groups of entries. The larger these groups, the more entries we can fix with a single programmatic approach. We want to find patterns that can be accurately described with a rule, such as "a digit, followed by a feet symbol, followed by one or two digits, followed by an inches symbol".

To look for such patterns, it helps to remove the entries that are consistent with being in inches and to view only the problematic entries. We thus write a function to automatically do this. We keep entries that either result in `NA`s when applying `as.numeric` or are outside a range of plausible heights. We permit a range that covers about 99.9999% of the adult population. We also use `suppressWarnings` to avoid the warning message we know `as.numeric` will gives us.

We apply this function and find the number of problematic entries:

```
problems <- reported_heights |>
  mutate(inches = suppressWarnings(as.numeric(height))) |>
  filter(is.na(inches) | inches < 50 | inches > 84) |>
  pull(height)
length(problems)
#> [1] 292
```

We can now view all the cases by simply printing them. If we do, we see that three patterns can be used to define three large groups within these exceptions.

1. A pattern of the form `x'y` or `x' y''` or `x'y"` with `x` and `y` representing feet and inches, respectively. Here are ten examples:

```
#> 5' 4" 5'7 5'7" 5'3" 5'11 5'9'' 5'10'' 5' 10 5'5" 5'2"
```

2. A pattern of the form `x.y` or `x,y` with `x` feet and `y` inches. Here are ten examples:

```
#> 5.3 5.5 6.5 5.8 5.6 5,3 5.9 6,8 5.5 6.2
```

3. Entries that were reported in centimeters rather than inches. Here are ten examples:

```
#> 150 175 177 178 163 175 178 165 165 180
```

Once we see these large groups following specific patterns, we can develop a plan of attack.

1. Convert entries fitting the first two patterns into one standardized one.
2. Leverage the standardization to extract the feet and inches and convert to inches.
3. Define a procedure for identifying entries that are in centimeters and convert them to inches.
4. Check again to see what entries were not fixed and see if we can tweak our approach to be more comprehensive.

At the end, we hope to have a script that makes web-based data collection methods robust to the most common user mistakes.

Remember that there is rarely just one way to perform these tasks. Here we pick one that helps us teach several useful techniques. But surely there is a more efficient way of performing the task.

To achieve our goal, we will use a technique that enables us to accurately detect patterns and extract the parts we want: *regular expressions* (regex). But first, we quickly describe how to *escape* the function of certain characters so that they can be included in strings.

16.3 Escaping

To define strings in R, we can use either double quotes or single quotes:

```
s <- "Hello!"
s <- 'Hello!'
```

ISTUDY

Make sure you choose the correct single quote, as opposed to the back quote `.

Now, what happens if the string we want to define includes double quotes? For example, if we want to write 10 inches like this `10"`? In this case you can't use:

```
s <- "10""
```

because this is just the string `10` followed by a double quote. If you type this into R, you get an error because you failed to close the double quote. To avoid this, we can use the single quotes:

```
s <- '10"'
```

If we print out `s` we see that the double quotes are *escaped* with the backslash `\`.

```
s
#> [1] "10\""
```

In fact, escaping with the backslash provides a way to define the string while still using the double quotes to define strings:

```
s <- "10\""
```

In R, the function `cat` lets us see what the string actually looks like:

```
cat(s)
#> 10"
```

Now, what if we want our string to be 5 feet written like this `5'`? In this case, we can use the double quotes or escape the single quote

```
s <- "5'"
s <- '5\''
```

So we've learned how to write 5 feet and 10 inches separately, but what if we want to write them together to represent *5 feet and 10 inches* like this `5'10"`? In this case, neither the single nor double quotes will work since `'5'10"'` closes the string after 5 and this `"5'10""` closes the string after 10. Keep in mind that if we type one of the above code snippets into R, it will get stuck waiting for you to close the open quote and you will have to exit the execution with the *esc* button.

To achieve the desired result we need to escape both quotes with the backslash `\`. You can escape either character that can be confused with a closing quote. These are the two options:

```
s <- '5\'10"'
s <- "5'10\""
```

Escaping characters is something we often have to use when processing strings. Another characters that often needs escaping is the backslash character itself. We can do this with `\\`. When using regular expression, the topic of the next section, we often have to escape the *special characters* used in this approach.

ISTUDY

16.4 Regular expressions

A regular expression (regex) is a way to describe specific patterns of characters of text. They can be used to determine if a given string matches the pattern. A set of rules has been defined to do this efficiently and precisely and here we show some examples. We can learn more about these rules by reading a detailed tutorials[1] [2]. The RStudio cheat sheet for **stringr** and regular expression[3] is also very useful.

The patterns supplied to the **stringr** functions can be a regex rather than a standard string. We will learn how this works through a series of examples.

Throughout this section you will see that we create strings to test out our regex. To do this, we define patterns that we know should match and also patterns that we know should not. We will call them `yes` and `no`, respectively. This permits us to check for the two types of errors: failing to match and incorrectly matching.

16.4.1 Strings are a regex

Technically any string is a regex, perhaps the simplest example is a single character. So the comma `,` used in the next code example is a simple example of searching with regex.

```
pattern <- ","
str_detect(c("1", "10", "100", "1,000", "10,000"), pattern)
#> [1] FALSE FALSE FALSE  TRUE  TRUE
```

Above, we noted that an entry included a `cm`. This is also a simple example of a regex. We can show all the entries that used `cm` like this:

```
str_subset(reported_heights$height, "cm")
#> [1] "165cm"  "170 cm"
```

16.4.2 Special characters

Now let's consider a slightly more complicated example. Which of the following strings contain the pattern `cm` or `inches`?

```
yes <- c("180 cm", "70 inches")
no <- c("180", "70''")
s <- c(yes, no)
```

[1] https://www.regular-expressions.info/tutorial.html
[2] http://r4ds.had.co.nz/strings.html#matching-patterns-with-regular-expressions
[3] https://rstudio.github.io/cheatsheets/strings.pdf

ISTUDY

We can do this with two searches:

```
str_detect(s, "cm") | str_detect(s, "inches")
#> [1]  TRUE  TRUE FALSE FALSE
```

However, we don't need to do this. The main feature that distinguishes the regex *language* from plain strings is that we can use special characters. These are characters with a meaning. We start by introducing `|` which means *or*. So if we want to know if either `cm` or `inches` appears in the strings, we can use the regex `cm|inches`:

```
str_detect(s, "cm|inches")
#> [1]  TRUE  TRUE FALSE FALSE
```

and obtain the correct answer.

Another special character that will be useful for identifying feet and inches values is `\d` which means any digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The backslash is used to distinguish it from the character `d`. In R, we have to *escape* the backslash `\` so we actually have to use `\\d` to represent digits. Here is an example:

```
yes <- c("5", "6", "5'10", "5 feet", "4'11")
no <- c("", ".", "Five", "six")
s <- c(yes, no)
pattern <- "\\d"
str_detect(s, pattern)
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

We take this opportunity to introduce the `str_view` function, which is helpful for troubleshooting as it shows us all matches for each string that has matches. Each match is surrounded by the characters `<` and `>`. Below, we see that `5` has one match and `5'10` has three matches.

```
str_view(s, pattern)
```

```
[1] | <5>
[2] | <6>
[3] | <5>'<1><0>
[4] | <5> feet
[5] | <4>'<1><1>
```

To view all strings, even if there wasn't a match, we can use the `match = NA` parameter.

```
str_view(s, pattern, match = NA)
```

ISTUDY

```
[1] | <5>
[2] | <6>
[3] | <5>'<1><0>
[4] | <5> feet
[5] | <4>'<1><1>
[6] |
[7] | .
[8] | Five
[9] | six
```

Another useful special character is `\w` which stands for *word character* and it matches any letter, number, or underscore.

There are many other special characters. We will learn some others below, but you can see most or all of them in the cheat sheet[4] mentioned earlier.

16.4.3 Character classes

Character classes are used to define a series of characters that can be matched. We define character classes with square brackets `[]`. So, for example, if we want the pattern to match only if we have a `5` or a `6`, we use the regex `[56]`:

```
str_view(s, "[56]", match = NA)
```

```
[1] | <5>
[2] | <6>
[3] | <5>'10
[4] | <5> feet
[5] | 4'11
[6] |
[7] | .
[8] | Five
[9] | six
```

Suppose we want to match values between 4 and 7. A common way to define character classes is with ranges. So, for example, `[0-9]` is equivalent to `\\d`. The pattern we want is therefore `[4-7]`.

```
yes <- as.character(4:7)
no <- as.character(1:3)
s <- c(yes, no)
str_detect(s, "[4-7]")
#> [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

[4] https://rstudio.github.io/cheatsheets/strings.pdf

ISTUDY

However, it is important to know that in regex everything is a character; there are no numbers. So `4` is the character `4` not the number four. Notice, for example, that `[1-20]` does **not** mean 1 through 20, it means the characters 1 through 2 or the character 0. So `[1-20]` simply means the character class composed of 0, 1, and 2.

Keep in mind that characters do have an order and the digits do follow the numeric order. So `0` comes before `1` which comes before `2` and so on. For the same reason, we can define lower case letters as `[a-z]`, upper case letters as `[A-Z]`, and `[a-zA-z]` as both.

Notice that `\w` is equivalent to `[a-zA-Z0-9_]`.

16.4.4 Anchors

What if we want a match when we have exactly 1 digit? This will be useful in our case study since feet are never more than 1 digit so a restriction will help us. One way to do this with regex is by using *anchors*, which let us define patterns that must start or end at a specific place. The two most common anchors are `^` and `$` which represent the beginning and end of a string, respectively. So the pattern `^\\d$` is read as "start of the string followed by one digit followed by end of string".

This pattern now only detects the strings with exactly one digit:

```
pattern <- "^\\d$"
yes <- c("1", "5", "9")
no <- c("12", "123", " 1", "a4", "b")
s <- c(yes, no)
str_view(s, pattern, match = NA)
```

```
[1] | <1>
[2] | <5>
[3] | <9>
[4] | 12
[5] | 123
[6] |  1
[7] | a4
[8] | b
```

The `1` does not match because it does not start with the digit but rather with a space, which is not easy to see.

16.4.5 Bounded quantifiers

For the inches part, we can have one or two digits. This can be specified in regex with *quantifiers*. This is done by following the pattern with curly brackets containing the number of times the previous entry can be repeated. We call the *bounded* because the numbers in the quantifier are limited by the numbers in the curly brackets. Later we learn about *unbounded quantifiers*.

ISTUDY

We use an example to illustrate. The pattern for one or two digits is:

```
pattern <- "^\\d{1,2}$"
yes <- c("1", "5", "9", "12")
no <- c("123", "a4", "b")
str_view(c(yes, no), pattern, match = NA)
```

```
[1] | <1>
[2] | <5>
[3] | <9>
[4] | <12>
[5] | 123
[6] | a4
[7] | b
```

In this case, `123` does **not** match, but `12` does. To look for our feet and inches pattern, we can add the symbols for feet `'` and inches `"` after the digits.

With what we have learned, we can now construct an example for the pattern `x'y"` with `x` feet and `y` inches.

```
pattern <- "^[4-7]'\\d{1,2}\"$"
```

The pattern is now getting complex, but you can look at it carefully and break it down:

- `^` = start of the string
- `[4-7]` = one digit, either 4,5,6 or 7
- `'` = feet symbol
- `\\d{1,2}` = one or two digits
- `\"` = inches symbol
- `$` = end of the string

Let's test it out:

```
yes <- c("5'7\"", "6'2\"",  "5'12\"")
no <- c("6,2\"", "6.2\"","I am 5'11\"", "3'2\"", "64")
str_detect(yes, pattern)
#> [1] TRUE TRUE TRUE
str_detect(no, pattern)
#> [1] FALSE FALSE FALSE FALSE FALSE
```

For now, we are permitting the inches to be 12 or larger. We will add a restriction later as the regex for this is a bit more complex than we are ready to show.

16.4.6 White space

Another problem we have is spaces. For example, our pattern does not match `5' 4"` because there is a space between `'` and `4` which our pattern does not permit. Spaces are characters and R does not ignore them:

ISTUDY

```
identical("Hi", "Hi ")
#> [1] FALSE
```

In regex, `\s` represents white space. To find patterns like `5' 4`, we can change our pattern to:

```
pattern_2 <- "^[4-7]'\\s\\d{1,2}\"$"
str_subset(problems, pattern_2)
#> [1] "5' 4\""  "5' 11\"" "5' 7\""
```

However, this will not match the patterns with no space. So do we need more than one regex pattern? It turns out we can use a quantifier for this as well.

16.4.7 Unbounded quantifiers: `*`, `?`, `+`

We want the pattern to permit spaces but not require them. Even if there are several spaces, like in this example `5'  4`, we still want it to match. There is a quantifier for exactly this purpose. In regex, the character `*` means zero or more instances of the previous character. Here is an example:

```
yes <- c("AB", "A1B", "A11B", "A111B", "A1111B")
no <- c("A2B", "A21B")
str_detect(yes, "A1*B")
#> [1] TRUE TRUE TRUE TRUE TRUE
str_detect(no, "A1*B")
#> [1] FALSE FALSE
```

The above matches the first string which has zero 1s and all the strings with one or more 1s. We can then improve our pattern by adding the `*` after the space character `\s`.

There are two other similar quantifiers. For none or once, we can use `?`, and for one or more, we can use `+`. You can see how they differ with this example:

```
data.frame(string = c("AB", "A1B", "A11B", "A111B", "A1111B"),
           none_or_more = str_detect(yes, "A1*B"),
           nore_or_once = str_detect(yes, "A1?B"),
           once_or_more = str_detect(yes, "A1+B"))
#>   string none_or_more nore_or_once once_or_more
#> 1     AB         TRUE         TRUE        FALSE
#> 2    A1B         TRUE         TRUE         TRUE
#> 3   A11B         TRUE        FALSE         TRUE
#> 4  A111B         TRUE        FALSE         TRUE
#> 5 A1111B         TRUE        FALSE         TRUE
```

We will actually use all three in our reported heights example, but we will see these in a later section.

ISTUDY

16.4.8 Not

To specify patterns that we do **not** want to detect, we can use the ^ symbol but only **inside** square brackets. Remember that outside the square bracket ^ means the start of the string. So, for example, if we want to detect digits that are preceded by anything except a letter we can do the following:

```
pattern <- "[^a-zA-Z]\\d"
yes <- c(".3", "+2", "-0","*4")
no <- c("A3", "B2", "C0", "E4")
str_detect(yes, pattern)
#> [1] TRUE TRUE TRUE TRUE
str_detect(no, pattern)
#> [1] FALSE FALSE FALSE FALSE
```

Another way to generate a pattern that searches for *everything except* is to use the upper case of the special character. For example `\\D` means anything other than a digit, `\\S` means anything except a space, and so on.

16.4.9 Groups

Groups are a powerful aspect of regex that permits the extraction of values. Groups are defined using parentheses. They don't affect the pattern matching per se. Instead, it permits tools to identify specific parts of the pattern so we can extract them.

We want to change heights written like `5.6` to `5'6`.

To avoid changing patterns such as `70.2`, we will require that the first digit be between 4 and 7 `[4-7]` and that the second be none or more digits `\\d*`. Let's start by defining a simple pattern that matches this:

```
pattern_without_groups <- "^[4-7],\\d*$"
```

We want to extract the digits so we can then form the new version using a period. These are our two groups, so we encapsulate them with parentheses:

```
pattern_with_groups <-  "^([4-7]),(\\d*)$"
```

We encapsulate the part of the pattern that matches the parts we want to keep for later use. Adding groups does not affect the detection, since it only signals that we want to save what is captured by the groups. Note that both patterns return the same result when using `str_detect`:

```
yes <- c("5,9", "5,11", "6,", "6,1")
no <- c("5'9", ",", "2,8", "6.1.1")
s <- c(yes, no)
str_detect(s, pattern_without_groups)
#> [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
str_detect(s, pattern_with_groups)
#> [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

ISTUDY

Once we define groups, we can use the function `str_match` to extract the values these groups define:

```
str_match(s, pattern_with_groups)
#>       [,1]   [,2] [,3]
#> [1,] "5,9"  "5"  "9"
#> [2,] "5,11" "5"  "11"
#> [3,] "6,"   "6"  ""
#> [4,] "6,1"  "6"  "1"
#> [5,] NA     NA   NA
#> [6,] NA     NA   NA
#> [7,] NA     NA   NA
#> [8,] NA     NA   NA
```

Notice that the second and third columns contain feet and inches, respectively. The first column is the part of the string matching the pattern. If no match occurred, we see an `NA`.

Now we can understand the difference between the functions `str_extract` and `str_match`. `str_extract` extracts only strings that match a pattern, not the values defined by groups:

```
str_extract(s, pattern_with_groups)
#> [1] "5,9"  "5,11" "6,"   "6,1"  NA     NA     NA     NA
```

16.4.10 Search and replace

Earlier we defined the object `problems` containing the strings that do not appear to be in inches. We can see that not too many of our problematic strings match the pattern:

```
pattern <- "^[4-7]'\\d{1,2}\"$"
sum(str_detect(problems, pattern))
#> [1] 14
```

To see why this is, we show some examples that expose why we don't have more matches:

```
problems[c(2, 10, 11, 12, 15)] |> str_view(pattern)
```

```
[2] | <5'7">
[3] | <5'3">
```

An initial problem we see immediately is that some students wrote out the words "feet" and "inches". We can see the entries that did this with the `str_subset` function:

```
str_subset(problems, "inches")
#> [1] "5 feet and 8.11 inches" "Five foot eight inches"
#> [3] "5 feet 7inches"         "5ft 9 inches"
#> [5] "5 ft 9 inches"          "5 feet 6 inches"
```

We also see that some entries used two single quotes `''` instead of a double quote `"`.

ISTUDY

```
str_subset(problems, "'")
#>  [1] "5'9''"   "5'10''"  "5'10''"  "5'3''"   "5'7''"   "5'6''"
#>  [7] "5'7.5''" "5'7.5''" "5'10''"  "5'11''"  "5'10''"  "5'5''"
```

To correct this, we can replace the different ways of representing inches and feet with a uniform symbol. We will use `'` for feet, whereas for inches we will simply not use a symbol since some entries were of the form `x'y`. Now, if we no longer use the inches symbol, we have to change our pattern accordingly:

```
pattern <- "^[4-7]'\\d{1,2}$"
```

If we do this replacement before the matching, we get many more matches:

```
problems |>
  str_replace("feet|ft|foot", "'") |> # replace feet, ft, foot with '
  str_replace("inches|in|''|\"", "") |> # remove all inches symbols
  str_detect(pattern) |>
  sum()
#> [1] 48
```

However, we still have many cases to go.

Note that in the code above, we leveraged the **stringr** consistency and used the pipe.

For now, we improve our pattern by adding `\\s*` in front of and after the feet symbol `'` to permit space between the feet symbol and the numbers. Now we match a few more entries:

```
pattern <- "^[4-7]\\s*'\\s*\\d{1,2}$"
problems |>
  str_replace("feet|ft|foot", "'") |> # replace feet, ft, foot with '
  str_replace("inches|in|''|\"", "") |> # remove all inches symbols
  str_detect(pattern) |>
  sum()
#> [1] 53
```

We might be tempted to avoid doing this by removing all the spaces with `str_replace_all`. However, when doing such an operation we need to make sure that it does not have unintended effects. In our reported heights examples, this will be a problem because some entries are of the form `x y` with space separating the feet from the inches. If we remove all spaces, we will incorrectly turn `x y` into `xy` which implies that a `6 1` would become `61` inches instead of `73` inches.

The second large type of problematic entries were of the form `x.y`, `x,y` and `x y`. We want to change all these to our common format `x'y`. But we can't just do a search and replace because we would change values such as `70.5` into `70'5`. Our strategy will therefore be to search for a very specific pattern that assures us feet and inches are being provided and then, for those that match, replace appropriately.

ISTUDY

16.4.11 Search and replace using groups

Another powerful aspect of groups is that you can refer to the extracted values in a regex when searching and replacing.

The regex special character for the `i`th group is `\\i`. So `\\1` is the value extracted from the first group, `\\2` the value from the second and so on. As a simple example, note that the following code will replace a comma with period, but only if it is after a digit between 4 and 7, and followed by zero or more digits:

```
pattern_with_groups <-  "^([4-7]),(\\d*)$"
yes <- c("5,9", "5,11", "6,", "6,1")
no <- c("5'9", ",", "2,8", "6.1.1")
s <- c(yes, no)
str_replace(s, pattern_with_groups, "\\1'\\2")
#> [1] "5'9"   "5'11"  "6'"    "6'1"   "5'9"   ","     "2,8"   "6.1.1"
```

We can use this to convert cases in our reported heights.

We are now ready to define a pattern that helps us convert all the `x.y`, `x,y`, and `x y` to our preferred format. We need to adapt `pattern_with_groups` to be a bit more flexible and capture all the cases.

```
pattern_with_groups <- "^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$"
```

Let's break this one down:

- `^` = start of the string
- `[4-7]` = one digit representing feet, either 4, 5, 6, or 7
- `\\s*` = none or more white space
- `[,\\.\\s+]` = feet and inches are separated by either `,`, `.` or at least one space
- `\\s*` = none or more white space
- `\\d*` = none or more digits representing inches
- `$` = end of the string

We can see that it appears to be working:

```
str_subset(problems, pattern_with_groups) |> head()
#> [1] "5.3"  "5.25" "5.5"  "6.5"  "5.8"  "5.6"
```

and will be able to perform the search and replace:

```
str_subset(problems, pattern_with_groups) |>
  str_replace(pattern_with_groups, "\\1'\\2") |> head()
#> [1] "5'3"  "5'25" "5'5"  "6'5"  "5'8"  "5'6"
```

Again, we will deal with the inches-larger-than-twelve challenge later.

16.4.12 Lookarounds

Lookarounds provide a way to ask for one or more conditions to be satisfied without moving the search forward or matching it. For example, you might want to check for multiple

ISTUDY

conditions and if they are matched, then return the pattern or part of the pattern that matched.

There are four types of lookarounds: lookahead `(?=pattern)`, lookbehind `(?<=pattern)`, negative lookahead `(?!pattern)`, and negative lookbehind `(?<!pattern)`.

The conventional example checking password that must satisfy several conditions such as 1) 8-16 word characters, 2) starts with a letter, and 3) has at least one digit. You can concatenate lookarounds to check for multiple conditions. For our example we can write

```
pattern <- "(?=\\w{8,16})(?=^[a-z|A-Z].*)(?=.*\\d+.*)"
```

A simpler example is changing all `superman` to `supergirl` without changing all the men to girl. We could use a lookaround like this:

```
s <- "Superman saved a man. The man thanked superman."
str_replace_all(s, "(?<=[Ss]uper)man", "girl")
#> [1] "Supergirl saved a man. The man thanked supergirl."
```

16.4.13 Separating variables

In Section 11.3 we introduced functions that can split columns into new ones. The `separate_wider_regex` uses regex instead of delimiters to separate variables in data frames. It uses an approach similar to regex groups and turns each of groups into a new column.

Suppose we have data frame like this:

```
tab <- data.frame(x = c("5'10", "6' 1", "5 ' 9", "5'11\""))
```

Note that using `separate_wider_delim` won't work here because the delimiter can varies across entries. With `separate_wider_regex` we can define flexible patterns that are matched to define each column.

```
patterns <- c(feet = "\\d", "\\s*'\\s*", inches = "\\d{1,2}", ".*")
tab |> separate_wider_regex(x, patterns = patterns)
#> # A tibble: 4 x 2
#>   feet  inches
#>   <chr> <chr>
#> 1 5     10
#> 2 6     1
#> 3 5     9
#> 4 5     11
```

By not naming the second and fourth entries of `patterns` we tell the function not to keep the values that match that pattern.

ISTUDY

16.5 Trimming

In general, spaces at the start or end of the string are uninformative. These can be particularly deceptive because sometimes they can be hard to see:

```
s <- "Hi "
cat(s)
#> Hi
identical(s, "Hi")
#> [1] FALSE
```

This is a general enough problem that there is a function dedicated to removing them: `str_trim`.

```
str_trim("5 ' 9 ")
#> [1] "5 ' 9"
```

16.6 Case conversion

Notice that regex is case sensitive. Often we want to match a word regardless of case. One approach to doing this is to first change everything to lower case and then proceeding ignoring case. As an example, note that one of the entries writes out numbers as words `Five foot eight inches`. Although not efficient, we could add 13 extra `str_replace` calls to convert `zero` to `0`, `one` to `1`, and so on. To avoid having to write two separate operations for `Zero` and `zero`, `One` and `one`, etc., we can use the `str_to_lower` function to make all works lower case first:

```
s <- c("Five feet eight inches")
str_to_lower(s)
#> [1] "five feet eight inches"
```

Other related functions are `str_to_upper` and `str_to_title`. We are now ready to define a procedure that converts all the problematic cases to inches.

16.7 Case study 1 continued: Putting it all together

We are now ready to put it all together and wrangle our reported heights data to try to recover as many heights as possible. The code is complex, but we will break it down into parts.

Let's see how many problematic entries we can recover with the approaches we covered in Section 16.4. We will first define a function that detects entries that are not reported as inches or centimeters:

ISTUDY

```
not_inches_or_cm <- function(x, smallest = 50, tallest = 84){
  inches <- suppressWarnings(as.numeric(x))
  is.na(inches) |
    !((inches >= smallest & inches <= tallest) |
        (inches/2.54 >= smallest & inches/2.54 <= tallest))
}
```

We can see how many entries are not inches or centimeters:

```
problems <- reported_heights |>
  filter(not_inches_or_cm(height)) |>
  pull(height)
length(problems)
#> [1] 200
```

Let's see how many feet'inches format we can recover applying the finding from Section 16.4. Specifically we replace feet/foot/ft with ', we remove the word inches and its abbreviations, then rewrite similar patterns into the desired feet'inches pattern. Once this is done we see what proportion don't fit the desired pattern

```
converted <- problems |>
  str_replace("feet|foot|ft", "'") |>
  str_remove_all("inches|in|''|\"") |>
  str_replace("^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$", "\\1'\\2")
index <- str_detect(converted, "^[4-7]\\s*'\\s*\\d{1,2}$")
mean(!index)
#> [1] 0.385
```

We see that a substantial proportion has not yet been fixed. Trial and error is a common approach to finding the regex pattern that satisfies all desired conditions. We can examine the remaining cases to try to decipher if new patterns we can use to fix them:

```
converted[!index]
#>  [1] "6"             "165cm"         "511"           "6"
#>  [5] "2"             ">9000"         "5 ' and 8.11 " "11111"
#>  [9] "6"             "103.2"         "19"            "5"
#> [13] "300"           "6'"            "6"             "Five ' eight "
#> [17] "7"             "214"           "6"             "0.7"
#> [21] "6"             "2'33"          "612"           "1,70"
#> [25] "87"            "5'7.5"         "5'7.5"         "111"
#> [29] "5' 7.78"       "12"            "6"             "yyy"
#> [33] "89"            "34"            "25"            "6"
#> [37] "6"             "22"            "684"           "6"
#> [41] "1"             "1"             "6*12"          "87"
#> [45] "6"             "1.6"           "120"           "120"
#> [49] "23"            "1.7"           "6"             "5"
#> [53] "69"            "5' 9 "         "5 ' 9 "        "6"
#> [57] "6"             "86"            "708,661"       "5 ' 6 "
#> [61] "6"             "649,606"       "10000"         "1"
```

ISTUDY

```
#> [65] "728,346"       "0"             "6"             "6"
#> [69] "6"             "100"           "88"            "6"
#> [73] "170 cm"        "7,283,465"     "5"             "5"
#> [77] "34"
```

We noticed that two students added `cm` and some entries have space at the end so we write incorporate this into our cleanup stage. We also notice that at least one entry wrote out numbers such as `Five foot eight inches`. We can use the `words()` function in the **english** package to change these to actual numbers. In preparation for our final product, we define a function that cleans up previously noticed problems and these new ones:

```
library(english)
cleanup <- function(s){
  s <- str_remove_all(s, "inches|in|''|\"|cm|and") |>
    str_trim() |>
    str_to_lower()
  for (i in 0:11) {
    s <- str_replace_all(s, words(i), as.character(i))
  }
  return(s)
}
```

Many also notice that students measuring exactly 5 or 6 feet did not enter any inches, for example `6'`, and our pattern requires that inches be included, and that some entries are in meters and some of these use European decimals, for example `1.6` and `1,70`. So we create a function that add these corrections to those already identified as needed to reformat the entry as feet'inches:

```
convert_format <- function(s){
  s |> str_replace("feet|foot|ft", "'") |>
    str_replace("^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$", "\\1'\\2") |>
    str_replace("^([56])'?$", "\\1'0") |>
    str_replace("^([12])\\s*,\\s*(\\d*)$", "\\1\\.\\2")
}
```

Now we are ready to wrangle our reported heights dataset. The strategy is as follows:

1. We start by defining a variable to keep a copy of the original entry, we then clean up and covert the `height` entry using the functions described above.
2. We then use the `separate_wider_regex_` function to extract feet and inches when the entry matches our feet'inches format.
3. We use a lookaround to make sure that entries that have numbers with no `'` following them don't match and return an `NA`.
4. Once this is done, we convert the feet and inches into inches.
5. Finally, we decide if the entries are in inches, centimeters, or meters and convert appropriately.

ISTUDY

```
patterns <- c(feet = "[4-7](?=\\s*'\\s*)",
              "\\s*'\\s*",
              inches = "\\d+\\.?\\d*")
smallest <- 50
tallest <- 84
new_heights <- reported_heights |>
  mutate(original = height,
         height = convert_format(cleanup(height))) |>
  separate_wider_regex(height, patterns = patterns,
                       too_few = "align_start",
                       cols_remove = FALSE) |>
  mutate(across(c(height, feet, inches), as.numeric)) |>
  mutate(guess = 12 * feet + inches) |>
  mutate(height = case_when(
   is.na(height) ~ as.numeric(NA),
   between(height, smallest, tallest) ~ height,  #inches
   between(height/2.54, smallest, tallest) ~ height/2.54, #cm
   between(height*100/2.54, smallest, tallest) ~ height*100/2.54, #meters
   TRUE ~ as.numeric(NA))) |>
  mutate(height = ifelse(is.na(height) &
                         inches <= 12 & between(guess, smallest, tallest),
                       guess, height)) |>
  select(-feet, -inches, -guess)
```

We see that we fix all but 44 entries. You can see that these are mostly un-fixable:

```
new_heights |> filter(is.na(height)) |> pull(original)
#>  [1] "511"       ">9000"     "5.25"      "11111"     "103.2"
#>  [6] "19"        "300"       "5.75"      "7"         "214"
#> [11] "0.7"       "2'33"      "612"       "87"        "111"
#> [16] "12"        "yyy"       "89"        "34"        "25"
#> [21] "22"        "684"       "1"         "1"         "6*12"
#> [26] "87"        "120"       "120"       "23"        "5.51"
#> [31] "5.69"      "86"        "708,661"   "5.25"      "649,606"
#> [36] "10000"     "1"         "728,346"   "0"         "100"
#> [41] "5.57"      "88"        "7,283,465" "34"
```

You can review the ones we did fix by typing:

```
new_heights |>
  filter(not_inches_or_cm(original)) |>
  select(original, height) |>
  arrange(height) |>
  View()
```

ISTUDY

A final observation is that if we look at the shortest students in our course:

```
new_heights |> arrange(height) |> head(n = 6)
#> # A tibble: 6 x 4
#>   time_stamp          sex    height original
#>   <chr>               <chr>   <dbl> <chr>
#> 1 2017-07-04 01:30:25 Male       50 50
#> 2 2017-09-07 10:40:35 Male       50 50
#> 3 2014-09-02 15:18:30 Female     51 51
#> 4 2016-06-05 14:07:20 Female     52 52
#> 5 2016-06-05 14:07:38 Female     52 52
#> # i 1 more row
```

We see heights of 50, 51, 52, and so on. These short heights are rare and it is likely that the students actually meant 5'0, 5'1, 5'2 and so on. Because we are not completely sure, we will leave them as reported. The object new_heights contains our final solution for this case study.

16.8 Case study 2: extracting tables from a PDF

One of the datasets provided in **dslabs** shows scientific funding rates by gender in the Netherlands:

```
library(dslabs)
research_funding_rates |>
  select("discipline", "success_rates_men", "success_rates_women")
#>            discipline success_rates_men success_rates_women
#> 1   Chemical sciences              26.5                25.6
#> 2   Physical sciences              19.3                23.1
#> 3             Physics              26.9                22.2
#> 4          Humanities              14.3                19.3
#> 5  Technical sciences              15.9                21.0
#> 6   Interdisciplinary              11.4                21.8
#> 7 Earth/life sciences              24.4                14.3
#> 8     Social sciences              15.3                11.5
#> 9    Medical sciences              18.8                11.2
```

The data comes from a paper published in the Proceedings of the National Academy of Science (PNAS)[5], a widely read scientific journal. However, the data is not provided in a spreadsheet; it is in a table in a PDF document. Here is a screenshot of the table:

[5] http://www.pnas.org/content/112/40/12349.abstract

ISTUDY

Table S1. Numbers of applications and awarded grants, along with success rates for male and female applicants, by scientific discipline

	Applications, *n*			Awards, *n*			Success rates, %		
Discipline	Total	Men	Women	Total	Men	Women	Total	Men	Women
Total	2,823	1,635	1,188	467	290	177	16.5	17.7_a	14.9_b
Chemical sciences	122	83	39	32	22	10	26.2	26.5_a	25.6_a
Physical sciences	174	135	39	35	26	9	20.1	19.3_a	23.1_a
Physics	76	67	9	20	18	2	26.3	26.9_a	22.2_a
Humanities	396	230	166	65	33	32	16.4	14.3_a	19.3_a
Technical sciences	251	189	62	43	30	13	17.1	15.9_a	21.0_a
Interdisciplinary	183	105	78	29	12	17	15.8	11.4_a	21.8_a
Earth/life sciences	282	156	126	56	38	18	19.9	24.4_a	14.3_b
Social sciences	834	425	409	112	65	47	13.4	15.3_a	11.5_a
Medical sciences	505	245	260	75	46	29	14.9	18.8_a	11.2_b

Success rates for male and female applicants with different subscripts differ reliably from one another ($P < 0.05$).

(Source: Romy van der Lee and Naomi Ellemers, PNAS 2015 112 (40) 12349–12353[6].)

We could extract the numbers by hand, but this could lead to human error. Instead, we can try to wrangle the data using R. We start by downloading the pdf document, then importing into R:

```
library("pdftools")
temp_file <- tempfile()
url <- paste0("https://web.archive.org/web/20150927033124/",
              "https://www.pnas.org/content/suppl/2015/09/16/",
              "1510159112.DCSupplemental/pnas.201510159SI.pdf")
download.file(url, temp_file, mode = "wb")
txt <- pdf_text(temp_file)
file.remove(temp_file)
```

> **Note**
>
> The `mode = "wb"` argument is only necessary if using Microsoft Windows. On macOS or Linux the default `mode = "w"` works. To understand this difference please refer to the `download.file` help file.

If we examine the object text, we notice that it is a character vector with an entry for each page. So we keep the page we want:

```
raw_data_research_funding_rates <- txt[2]
```

The steps above can actually be skipped because we include this raw data in the **dslabs** package as well.

[6] http://www.pnas.org/content/112/40/12349

ISTUDY

Examining the object `raw_data_research_funding_rates` we see that it is a long string and each line on the page, including the table rows, are separated by the symbol for newline: `\n`. A first step in converting this into a data frame is to store rows separately in a way that will make them easy to access. For this we use the `str_split` function:

```
tab <- str_split(raw_data_research_funding_rates, "\n+")
```

> **Note**
>
> On macOS or Linux you can simply use `\n` as the separator. Microsoft Windows and macOS (which is based on Unix) use different conventions for line endings in text files and a result `raw_data_research_funding_rates` has more `\n` when using Windows.

Because we start off with just one string, we end up with a list with just one entry.

```
tab <- tab[[1]]
```

By examining `tab` we see that the information for the column names is the third and fourth entries:

```
the_names_1 <- tab[3]
the_names_2 <- tab[4]
```

The first of these rows looks like this:

```
#>                                                  Applications, n
#>                    Awards, n                  Success rates, %
```

We want to create one vector with one name for each column. Using some of the functions we have just learned, we do this.

Let's start with `the_names_1`, shown above. We want to remove the leading space and anything following the comma. We use regex for the latter. Then we can obtain the elements by splitting strings separated by space. We want to split only when there are 2 or more spaces to avoid splitting `Success rates`. So we use the regex `\\s{2,}`

```
the_names_1 <- the_names_1 |>
  str_trim() |>
  str_replace_all(",\\s.", "") |>
  str_split("\\s{2,}", simplify = TRUE)
the_names_1
#>      [,1]           [,2]     [,3]
#> [1,] "Applications" "Awards" "Success rates"
```

Now we will look at `the_names_2`:

```
#>                          Discipline              Total     Men      Women
#> n         Total    Men       Women          Total    Men      Women
```

ISTUDY

Here we want to trim the leading space and then split by space as we did for the first line:

```
the_names_2 <- the_names_2 |>
  str_trim() |>
  str_split("\\s+", simplify = TRUE)
the_names_2
#>      [,1]          [,2]    [,3]  [,4]    [,5]    [,6]  [,7]    [,8]
#> [1,] "Discipline" "Total" "Men" "Women" "Total" "Men" "Women" "Total"
#>      [,9]  [,10]
#> [1,] "Men" "Women"
```

We can then join these to generate one name for each column:

```
tmp_names <- paste(rep(the_names_1, each = 3), the_names_2[-1], sep = "_")
the_names <- c(the_names_2[1], tmp_names) |>
  str_to_lower() |>
  str_replace_all("\\s", "_")
the_names
#>  [1] "discipline"          "applications_total"  "applications_men"
#>  [4] "applications_women"  "awards_total"        "awards_men"
#>  [7] "awards_women"        "success_rates_total" "success_rates_men"
#> [10] "success_rates_women"
```

Now we are ready to get the actual data. By examining the `tab` object, we notice that the information is in lines 6 through 14. We can use `str_split` again to achieve our goal:

```
new_research_funding_rates <- tab[6:14] |>
  str_trim() |>
  str_split("\\s{2,}", simplify = TRUE) |>
  data.frame() |>
  setNames(the_names) |>
  mutate(across(-1, parse_number))
new_research_funding_rates |> as_tibble()
#> # A tibble: 9 x 10
#>   discipline      applications_total applications_men applications_women
#>   <chr>                       <dbl>            <dbl>              <dbl>
#> 1 Chemical scien~               122               83                 39
#> 2 Physical scien~               174              135                 39
#> 3 Physics                        76               67                  9
#> 4 Humanities                    396              230                166
#> 5 Technical scie~               251              189                 62
#> # i 4 more rows
#> # i 6 more variables: awards_total <dbl>, awards_men <dbl>,
#> #   awards_women <dbl>, success_rates_total <dbl>,
#> #   success_rates_men <dbl>, success_rates_women <dbl>
```

We can see that the objects are identical:

```
identical(research_funding_rates, new_research_funding_rates)
#> [1] TRUE
```

ISTUDY

16.9 Renaming levels

Another common operation involving strings is renaming the levels of of a categorical variables. Let's say you have really long names for your levels. If you will be displaying them in plots, you might want to use shorter versions of these names. For example, in character vectors with country names, you might want to change "United States of America" to "USA" and "United Kingdom" to UK, and so on.

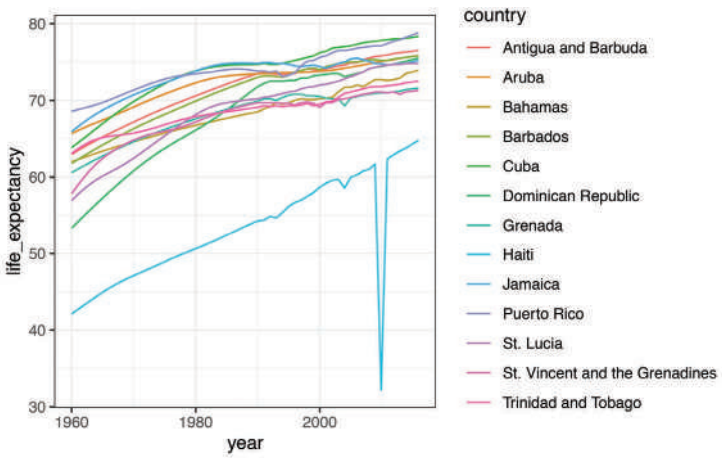
Rather than changing each entry, a more efficient approach is to change the levels.

Here is an example that shows how to rename countries with long names:

```
library(dslabs)
```

Suppose we want to show life expectancy time series by country for the Caribbean. If we make this plot

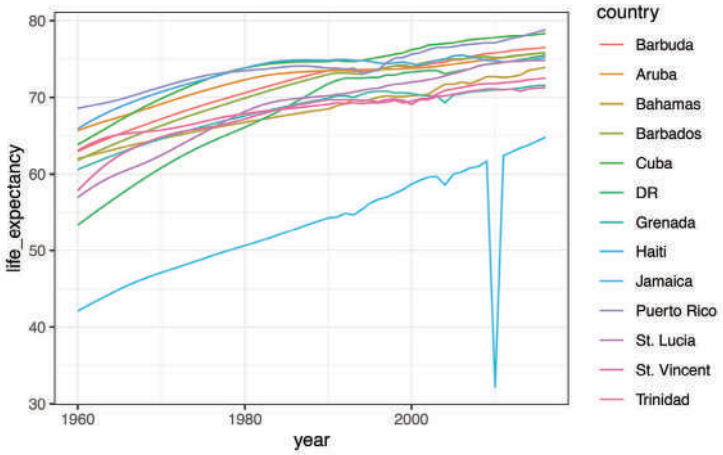
```
gapminder |>
  filter(region == "Caribbean") |>
  ggplot(aes(year, life_expectancy, color = country)) +
  geom_line()
```



we see that the legend takes up much of the plot because we have four countries with names longer than 12 characters. We can rename these levels using the `case_when` function:

```
x <- levels(gapminder$country)
levels(gapminder$country) <- case_when(
  x == "Antigua and Barbuda" ~ "Barbuda",
  x == "Dominican Republic" ~ "DR",
  x == "St. Vincent and the Grenadines" ~ "St. Vincent",
  x == "Trinidad and Tobago" ~ "Trinidad",
  .default = x)
```

```
gapminder |>
  filter(region == "Caribbean") |>
  ggplot(aes(year, life_expectancy, color = country)) +
  geom_line()
```



We can instead use the `fct_recode` function in the **forcats** package:

```
library(forcats)
gapminder$country <-
  fct_recode(gapminder$country,
             "Barbuda" = "Antigua and Barbuda",
             "DR" = "Dominican Republic",
             "St. Vincent" = "St. Vincent and the Grenadines",
             "Trinidad" = "Trinidad and Tobago")
```

16.10 Exercises

1. Complete all lessons and exercises in the RegexOne[7] online interactive tutorial.
2. In the `extdata` directory of the **dslabs** package, you will find a PDF file containing daily mortality data for Puerto Rico from Jan 1, 2015 to May 31, 2018. You can find the file like this:

```
fn <- system.file("extdata", "RD-Mortality-Report_2015-18-180531.pdf",
                  package="dslabs")
```

[7] https://regexone.com/

ISTUDY

Find and open the file or open it directly from RStudio. On a Mac, you can type:

```
system2("open", args = fn)
```

and on Windows, you can type:

```
system("cmd.exe", input = paste("start", fn))
```

Which of the following best describes this file:

(a) It is a table. Extracting the data will be easy.
(b) It is a report written in prose. Extracting the data will be impossible.
(c) It is a report combining graphs and tables. Extracting the data seems possible.
(d) It shows graphs of the data. Extracting the data will be difficult.

3. We are going to create a tidy dataset with each row representing one observation. The variables in this dataset will be year, month, day, and deaths. Start by installing and loading the **pdftools** package:

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

Now read-in `fn` using the `pdf_text` function and store the results in an object called `txt`. Which of the following best describes what you see in `txt`?

(a) A table with the mortality data.
(b) A character string of length 12. Each entry represents the text in each page. The mortality data is in there somewhere.
(c) A character string with one entry containing all the information in the PDF file.
(d) An html document.

4. Extract the ninth page of the PDF file from the object `txt`, then use the `str_split` from the **stringr** package so that you have each line in a different entry. Call this string vector `s`. Then look at the result and choose the one that best describes what you see.

(a) It is an empty string.
(b) I can see the figure shown in page 1.
(c) It is a tidy table.
(d) I can see the table! But there is a bunch of other stuff we need to get rid of.

5. What kind of object is `s` and how many entries does it have?

6. We see that the output is a list with one component. Redefine `s` to be the first entry of the list. What kind of object is `s` and how many entries does it have?

7. When inspecting the string we obtained above, we see a common problem: white space before and after the other characters. Trimming is a common first step in string processing. These extra spaces will eventually make splitting the strings hard so we start by removing them. We learned about the command `str_trim` that removes spaces at the start or end of the strings. Use this function to trim `s`.

ISTUDY

8. We want to extract the numbers from the strings stored in `s`. However, there are many non-numeric characters that will get in the way. We can remove these, but before doing this we want to preserve the string with the column header, which includes the month abbreviation. Use the `str_which` function to find the rows with a header. Save these results to `header_index`. Hint: find the first string that matches the pattern `2015` using the `str_which` function.

9. Now we are going to define two objects: `month` will store the month and `header` will store the column names. Identify which row contains the header of the table. Save the content of the row into an object called `header`, then use `str_split` to help define the two objects we need. Hints: the separator here is one or more spaces. Also, consider using the `simplify` argument.

10. Notice that toward the end of the page you see a *totals* row followed by rows with other summary statistics. Create an object called `tail_index` with the index of the *totals* entry.

11. Because our PDF page includes graphs with numbers, some of our rows have just one number (from the y-axis of the plot). Use the `str_count` function to create an object `n` with the number of numbers in each each row. Hint: you can write a regex for number like this `\\d+`.

12. We are now ready to remove entries from rows that we know we don't need. The entry `header_index` and everything before it should be removed. Entries for which `n` is 1 should also be removed, and the entry `tail_index` and everything that comes after it should be removed as well.

13. Now we are ready to remove all the non-numeric entries. Do this using regex and the `str_remove_all` function. Hint: remember that in regex, using the upper case version of a special character usually means the opposite. So `\\D` means "not a digit". Remember you also want to keep spaces.

14. To convert the strings into a table, use the `str_split_fixed` function. Convert `s` into a data matrix with just the day and death count data. Hints: note that the separator is one or more spaces. Make the argument `n` a value that limits the number of columns to the values in the 4 columns and the last column captures all the extra stuff. Then keep only the first four columns.

15. Now you are almost ready to finish. Add column names to the matrix, including one called `day`. Also, add a column with the month. Call the resulting object `dat`. Finally, make sure the day is an integer not a character. Hint: use only the first five columns.

16. Now finish it up by tidying `tab` with the `pivot_longer` function.

17. Make a plot of deaths versus day with color to denote year. Exclude 2018 since we do not have data for the entire year.

18. Now that we have wrangled this data step-by-step, put it all together in one R chunk, using the pipe as much as possible. Hint: first define the indexes, then write one line of code that does all the string processing.

19. Advanced: let's return to the MLB Payroll example from the web scraping section. Use what you have learned in the web scraping and string processing chapters to extract the payroll for the New York Yankees, Boston Red Sox, and Oakland A's and plot them as a function of time.

ISTUDY

17

Text analysis

With the exception of labels used to represent categorical data, we have focused on numerical data. But in many applications, data starts as text. Well-known examples are spam filtering, cyber-crime prevention, counter-terrorism, and sentiment analysis. In all these cases, the raw data is composed of free-form text. Our task is to extract insights from these data. In this section, we learn how to generate useful numerical summaries from text data to which we can apply some of the powerful data visualization and analysis techniques we have learned.

17.1 Case study: Trump tweets

During the 2016 US presidential election, then candidate Donald J. Trump used his twitter account as a way to communicate with potential voters. On August 6, 2016, Todd Vaziri tweeted[1] about Trump that "Every non-hyperbolic tweet is from iPhone (his staff). Every hyperbolic tweet is from Android (from him)." David Robinson conducted an analysis[2] to determine if data supported this assertion. Here, we go through David's analysis to learn some of the basics of text analysis. To learn more about text analysis in R, we recommend the Text Mining with R book[3] by Julia Silge and David Robinson.

We will use the following libraries:

```
library(tidyverse)
library(scales)
library(tidytext)
library(textdata)
library(dslabs)
```

X, formerly known as twitter, provides an API that permits downloading tweets. Brendan Brown runs the trump archive[4], which compiles tweet data from Trump's account. The **dslabs** package includes tweets from the following range:

```
range(trump_tweets$created_at)
#> [1] "2009-05-04 13:54:25 EST" "2018-01-01 08:37:52 EST"
```

[1] https://twitter.com/tvaziri/status/762005541388378112/photo/1
[2] http://varianceexplained.org/r/trump-tweets/
[3] https://www.tidytextmining.com/
[4] https://www.thetrumparchive.com/

DOI: 10.1201/9781003220923-17

The data frame includes the the following variables:

```
names(trump_tweets)
#> [1] "source"                  "id_str"
#> [3] "text"                    "created_at"
#> [5] "retweet_count"           "in_reply_to_user_id_str"
#> [7] "favorite_count"          "is_retweet"
```

The help file `?trump_tweets` provides details on what each variable represents. The actual tweets are in the `text` variable:

```
trump_tweets$text[16413] |> str_wrap(width = options()$width) |> cat()
#> Great to be back in Iowa! #TBT with @JerryJrFalwell joining me in
#> Davenport- this past winter. #MAGA https://t.co/A5IF0QHnic
```

and the source variable tells us which device was used to compose and upload each tweet:

```
trump_tweets |> count(source) |> arrange(desc(n)) |> head(5)
#>                source     n
#> 1  Twitter Web Client 10718
#> 2 Twitter for Android  4652
#> 3  Twitter for iPhone  3962
#> 4           TweetDeck   468
#> 5     TwitLonger Beta   288
```

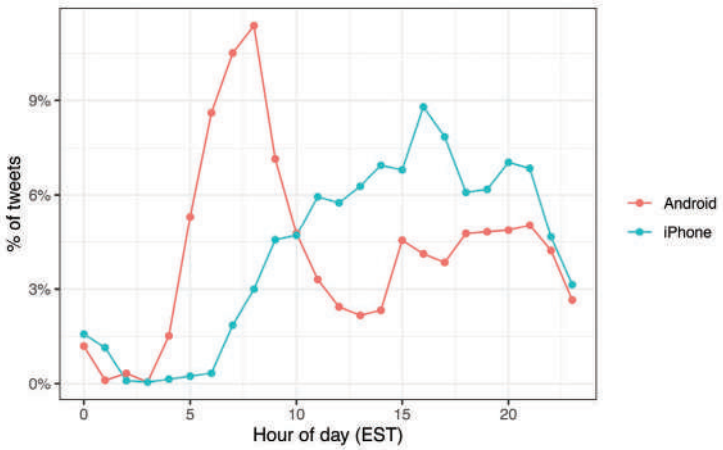
We are interested in what happened during the 2016 campaign, so for this analysis we will focus on what was tweeted between the day Trump announced his campaign and election day. We define the following table containing just the tweets from that time period. We remove the `Twitter for` part of the source, only keep tweets from Android or iPhone, and filter out retweets.

```
campaign_tweets <- trump_tweets |>
  filter(source %in% paste("Twitter for", c("Android", "iPhone")) &
           created_at >= ymd("2015-06-17") &
           created_at < ymd("2016-11-08")) |>
  mutate(source = str_remove(source, "Twitter for ")) |>
  filter(!is_retweet) |>
  arrange(created_at) |>
  as_tibble()
```

We can now use data visualization to explore the possibility that two different groups were tweeting from these devices. For each tweet, we will extract the hour, Eastern Standard Time (EST), it was tweeted and then compute the proportion of tweets tweeted at each hour for each device:

ISTUDY

```
campaign_tweets |>
  mutate(hour = hour(with_tz(created_at, "EST"))) |>
  count(source, hour) |>
  group_by(source) |>
  mutate(percent = n / sum(n)) |>
  ungroup() |>
  ggplot(aes(hour, percent, color = source)) +
  geom_line() +
  geom_point() +
  scale_y_continuous(labels = percent_format()) +
  labs(x = "Hour of day (EST)", y = "% of tweets", color = "")
```



We notice a big peak for the Android in the early hours of the morning, between 6 and 8 AM. There seems to be a clear difference in these patterns. We will therefore assume that two different entities are using these two devices.

We will now study how the text of the tweets differ when we compare Android to iPhone. To do this, we introduce the **tidytext** package.

17.2 Text as data

The **tidytext** package helps us convert free-form text into a tidy table. Having the data in this format greatly facilitates data visualization and the use of statistical techniques.

The main function needed to achieve this is `unnest_tokens`. A *token* refers to a unit that we are considering to be a data point. The most common *token* will be words, but they can also be single characters, n-grams, sentences, lines, or a pattern defined by a regex. The function will take a vector of strings and extract the tokens so that each one gets a row in the new table. Here is a simple example:

ISTUDY

```
poem <- c("Roses are red,", "Violets are blue,",
          "Sugar is sweet,", "And so are you.")
example <- tibble(line = c(1, 2, 3, 4),
                  text = poem)
example
#> # A tibble: 4 x 2
#>    line text
#>   <dbl> <chr>
#> 1     1 Roses are red,
#> 2     2 Violets are blue,
#> 3     3 Sugar is sweet,
#> 4     4 And so are you.
example |> unnest_tokens(word, text)
#> # A tibble: 13 x 2
#>    line word
#>   <dbl> <chr>
#> 1     1 roses
#> 2     1 are
#> 3     1 red
#> 4     2 violets
#> 5     2 are
#> # i 8 more rows
```

Now let's look at Trump's tweets. We will look at tweet number 3008 because it will later permit us to illustrate a couple of points:

```
i <- 3008
campaign_tweets$text[i] |> str_wrap(width = 65) |> cat()
#> Great to be back in Iowa! #TBT with @JerryJrFalwell joining me in
#> Davenport- this past winter. #MAGA https://t.co/A5IF0QHnic
campaign_tweets[i,] |>
  unnest_tokens(word, text) |>
  pull(word)
#>  [1] "great"          "to"             "be"             "back"
#>  [5] "in"             "iowa"           "tbt"            "with"
#>  [9] "jerryjrfalwell" "joining"        "me"             "in"
#> [13] "davenport"      "this"           "past"           "winter"
#> [17] "maga"           "https"          "t.co"           "a5if0qhnic"
```

Note that the function tries to convert tokens into words. A minor adjustment is to remove the links to pictures:

ISTUDY

```
links_to_pics <- "https://t.co/[A-Za-z\\d]+|&amp;"
campaign_tweets[i,] |>
  mutate(text = str_remove_all(text, links_to_pics))  |>
  unnest_tokens(word, text) |>
  pull(word)
#>  [1] "great"          "to"             "be"             "back"
#>  [5] "in"             "iowa"           "tbt"            "with"
#>  [9] "jerryjrfalwell" "joining"        "me"             "in"
#> [13] "davenport"      "this"           "past"           "winter"
#> [17] "maga"
```

Now we are now ready to extract the words from all our tweets.

```
tweet_words <- campaign_tweets |>
  mutate(text = str_remove_all(text, links_to_pics))  |>
  unnest_tokens(word, text)
```

And we can now answer questions such as "what are the most commonly used words?":

```
tweet_words |>
  count(word) |>
  arrange(desc(n))
#> # A tibble: 6,264 x 2
#>   word      n
#>   <chr> <int>
#> 1 the    2330
#> 2 to     1413
#> 3 and    1245
#> 4 in     1190
#> 5 i      1151
#> # i 6,259 more rows
```

It is not surprising that these are the top words, which are not informative. The *tidytext* package has a database of these commonly used words, referred to as *stop words*, in text analysis:

```
head(stop_words)
#> # A tibble: 6 x 2
#>   word  lexicon
#>   <chr> <chr>
#> 1 a     SMART
#> 2 a's   SMART
#> 3 able  SMART
#> 4 about SMART
#> 5 above SMART
#> # i 1 more row
```

ISTUDY

If we filter out rows representing stop words with `filter(!word %in% stop_words$word)`:

```
tweet_words <- campaign_tweets |>
  mutate(text = str_remove_all(text, links_to_pics))  |>
  unnest_tokens(word, text) |>
  filter(!word %in% stop_words$word )
```

we end up with a much more informative set of top 10 tweeted words:

```
tweet_words |>
  count(word) |>
  top_n(10, n) |>
  arrange(desc(n))
#> # A tibble: 10 x 2
#>   word                      n
#>   <chr>                 <int>
#> 1 trump2016               415
#> 2 hillary                 407
#> 3 people                  304
#> 4 makeamericagreatagain   298
#> 5 america                 255
#> # i 5 more rows
```

Some exploration of the resulting words (not shown here) reveals a couple of unwanted characteristics in our tokens. First, some of our tokens are just numbers (years, for example). We want to remove these and we can find them using the regex `^\d+$`. Second, some of our tokens come from a quote and they start with `'`. We want to remove the `'` when it is at the start of a word so we will just `str_replace`. We add these two lines to the code above to generate our final table:

```
tweet_words <- campaign_tweets |>
  mutate(text = str_remove_all(text, links_to_pics))  |>
  unnest_tokens(word, text) |>
  filter(!word %in% stop_words$word &
           !str_detect(word, "^\\d+$")) |>
  mutate(word = str_replace(word, "^'", ""))
```

Now that we have all our words in a table, along with information about what device was used to compose the tweet they came from, we can start exploring which words are more common when comparing Android to iPhone.

For each word, we want to know if it is more likely to come from an Android tweet or an iPhone tweet. We therefore compute, for each word, its frequency among words tweeted from Android and iPhone, respectively.

```
android_vs_iphone <- tweet_words |>
 count(word, source) |>
 pivot_wider(names_from = "source", values_from = "n", values_fill = 0)|>
 mutate(p_a = Android / sum(Android), p_i = iPhone / sum(iPhone),
        percent_diff = (p_a - p_i) / ((p_a + p_i)/2) * 100)
```

ISTUDY

For words appearing at least 100 times in total, here are the highest percent differences for Android

```
android_vs_iphone |> filter(Android + iPhone >= 100) |>
  arrange(desc(percent_diff))
#> # A tibble: 30 x 6
#>   word        Android iPhone     p_a     p_i percent_diff
#>   <chr>         <int>  <int>   <dbl>   <dbl>        <dbl>
#> 1 bad             104     26 0.00645 0.00188        110.
#> 2 crooked         156     49 0.00968 0.00354         92.9
#> 3 cnn             116     37 0.00720 0.00267         91.7
#> 4 ted              86     28 0.00533 0.00202         90.1
#> 5 interviewed      76     25 0.00471 0.00180         89.3
#> # i 25 more rows
```

and the top for iPhone:

```
android_vs_iphone |> filter(Android + iPhone >= 100) |>
  arrange(percent_diff)
#> # A tibble: 30 x 6
#>   word                  Android iPhone       p_a     p_i percent_diff
#>   <chr>                   <int>  <int>     <dbl>   <dbl>        <dbl>
#> 1 makeamericagreatagain       0    298 0         0.0215        -200
#> 2 join                        1    157 0.0000620 0.0113        -198.
#> 3 trump2016                   3    412 0.000186  0.0297        -198.
#> 4 tomorrow                   24    101 0.00149   0.00729       -132.
#> 5 vote                       46     67 0.00285   0.00484        -51.6
#> # i 25 more rows
```

We already see somewhat of a pattern in the types of words that are being tweeted more from one device versus the other. However, we are not interested in specific words but rather in the tone. Vaziri's assertion is that the Android tweets are more hyperbolic. So how can we check this with data? *Hyperbolic* is a hard sentiment to extract from words as it relies on interpreting phrases. However, words can be associated to more basic sentiment such as anger, fear, joy, and surprise. In the next section, we demonstrate basic sentiment analysis.

17.3 Sentiment analysis

In sentiment analysis, we assign a word to one or more "sentiments". Although this approach will miss context-dependent sentiments, such as sarcasm, when performed on large numbers of words, summaries can provide insights.

The first step in sentiment analysis is to assign a sentiment to each word. As we demonstrate, the **tidytext** package includes several maps or lexicons. The **textdata** package includes several of these lexicons.

The `bing` lexicon divides words into `positive` and `negative` sentiments. We can see this using the *tidytext* function `get_sentiments`:

ISTUDY

```
get_sentiments("bing")
```

The `AFINN` lexicon assigns a score between -5 and 5, with -5 the most negative and 5 the most positive. Note that this lexicon needs to be downloaded the first time you call the function `get_sentiment`:

```
get_sentiments("afinn")
```

The `loughran` and `nrc` lexicons provide several different sentiments. Note that these also have to be downloaded the first time you use them.

```
get_sentiments("loughran") |> count(sentiment)
#> # A tibble: 6 x 2
#>   sentiment        n
#>   <chr>        <int>
#> 1 constraining   184
#> 2 litigious      904
#> 3 negative      2355
#> 4 positive       354
#> 5 superfluous     56
#> # i 1 more row
```

```
get_sentiments("nrc") |> count(sentiment)
#> # A tibble: 10 x 2
#>   sentiment        n
#>   <chr>        <int>
#> 1 anger         1245
#> 2 anticipation   837
#> 3 disgust       1056
#> 4 fear          1474
#> 5 joy            687
#> # i 5 more rows
```

For our analysis, we are interested in exploring the different sentiments of each tweet so we will use the `nrc` lexicon:

```
nrc <- get_sentiments("nrc") |>
  select(word, sentiment)
```

We can combine the words and sentiments using `inner_join`, which will only keep words associated with a sentiment. Here are 5 random words extracted from the tweets:

```
tweet_words |> inner_join(nrc, by = "word",
  relationship = "many-to-many") |>
  select(source, word, sentiment) |>
  sample_n(5)
#> # A tibble: 5 x 3
#>   source  word     sentiment
#>   <chr>   <chr>    <chr>
```

ISTUDY

```
#> 1 Android enjoy    joy
#> 2 iPhone  terrific sadness
#> 3 iPhone  tactics  trust
#> 4 Android clue     anticipation
#> 5 iPhone  change   fear
```

> **Note**
>
> `relationship = "many-to-many"` is added to address a warning that arises from `left_join` detecting an "unexpected many-to-many relationship". However, this behavior is actually expected in this context because many words have multiple sentiments associated with them.

Now we are ready to perform a quantitative analysis comparing Android and iPhone by comparing the sentiments of the tweets posted from each device. Here we could perform a tweet-by-tweet analysis, assigning a sentiment to each tweet. However, this will be challenging since each tweet will have several sentiments attached to it, one for each word appearing in the lexicon. For illustrative purposes, we will perform a much simpler analysis: we will count and compare the frequencies of each sentiment appearing in each device.

```
sentiment_counts <- tweet_words |>
  left_join(nrc, by = "word", relationship = "many-to-many") |>
  count(source, sentiment) |>
  pivot_wider(names_from = "source", values_from = "n") |>
  mutate(sentiment = replace_na(sentiment, replace = "none"))
sentiment_counts
#> # A tibble: 11 x 3
#>   sentiment    Android iPhone
#>   <chr>          <int>  <int>
#> 1 anger            962    527
#> 2 anticipation     917    707
#> 3 disgust          639    314
#> 4 fear             799    486
#> 5 joy              695    536
#> # i 6 more rows
```

For each sentiment, we can compute the percent difference in proportion for Android compared to iPhone:

```
sentiment_counts |>
  mutate(p_a = Android / sum(Android) ,
         p_i = iPhone / sum(iPhone),
         percent_diff = (p_a - p_i) / ((p_a + p_i)/2) * 100) |>
  arrange(desc(percent_diff))
#> # A tibble: 11 x 6
#>   sentiment Android iPhone    p_a    p_i percent_diff
#>   <chr>       <int>  <int>  <dbl>  <dbl>        <dbl>
#> 1 disgust       639    314 0.0290 0.0178         48.1
#> 2 anger         962    527 0.0437 0.0298         37.8
```

ISTUDY

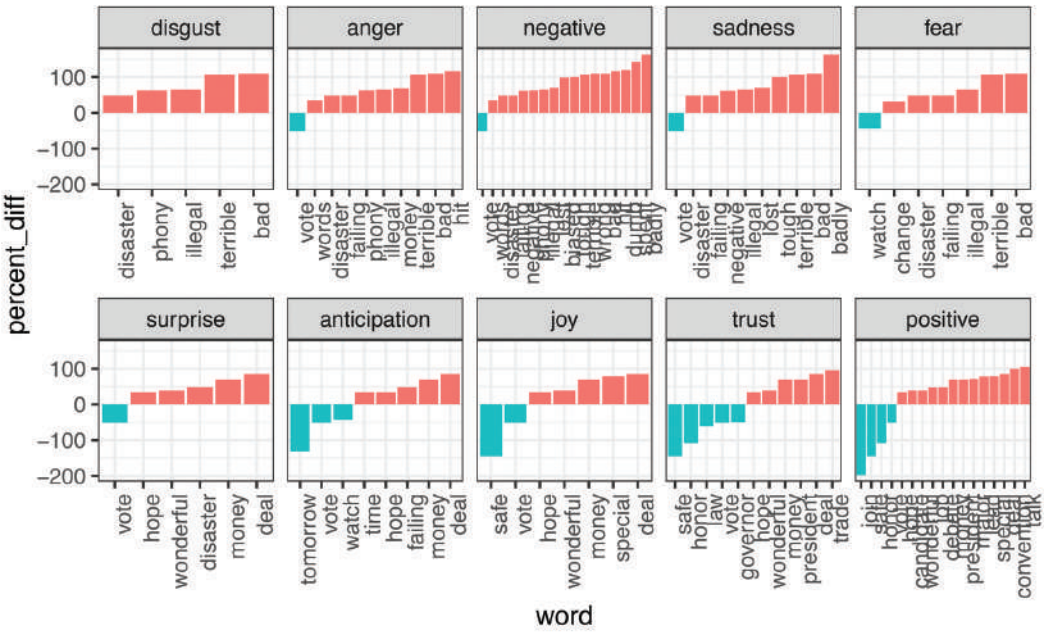
```
#> 3 negative     1657    931 0.0753 0.0527          35.3
#> 4 sadness       901    514 0.0409 0.0291          33.8
#> 5 fear          799    486 0.0363 0.0275          27.6
#> # i 6 more rows
```

So we do see some differences and the order is interesting: the largest three sentiments are disgust, anger, and negative!

If we are interested in exploring which specific words are driving these differences, we can refer back to our `android_vs_iphone` object:

```
android_vs_iphone |> inner_join(nrc, by = "word") |>
  filter(sentiment == "disgust") |>
  arrange(desc(percent_diff))
#> # A tibble: 157 x 7
#>   word       Android iPhone       p_a   p_i percent_diff sentiment
#>   <chr>        <int>  <int>     <dbl> <dbl>        <dbl> <chr>
#> 1 abuse            1      0 0.0000620     0          200 disgust
#> 2 angry           10      0 0.000620      0          200 disgust
#> 3 arrogant         2      0 0.000124      0          200 disgust
#> 4 attacking        5      0 0.000310      0          200 disgust
#> 5 belittle         2      0 0.000124      0          200 disgust
#> # i 152 more rows
```

and we can make a graph:



This is just a simple example of the many analyses one can perform with tidytext. To learn more, we again recommend the Tidy Text Mining book[5].

17.4 Exercises

Project Gutenberg is a digital archive of public domain books. The R package **gutenbergr** facilitates the importation of these texts into R.

You can install and load by typing:

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

You can see the books that are available like this:

```
gutenberg_metadata
```

1. Use `str_detect` to find the ID of the novel *Pride and Prejudice*.
2. We notice that there are several versions. The `gutenberg_works()` function filters this table to remove replicates and include only English language works. Read the help file and use this function to find the ID for *Pride and Prejudice*.
3. Use the `gutenberg_download` function to download the text for Pride and Prejudice. Save it to an object called `book`.
4. Use the **tidytext** package to create a tidy table with all the words in the text. Save the table in an object called `words`
5. We will later make a plot of sentiment versus location in the book. For this, it will be useful to add a column with the word number to the table.
6. Remove the stop words and numbers from the `words` object. Hint: use the `anti_join`.
7. Now use the `AFINN` lexicon to assign a sentiment value to each word.
8. Make a plot of sentiment score versus location in the book and add a smoother.
9. Assume there are 300 words per page. Convert the locations to pages and then compute the average sentiment in each page. Plot that average score by page. Add a smoother that appears to go through data.

[5] https://www.tidytextmining.com/

ISTUDY

Part IV

Productivity Tools

ISTUDY



Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

ISTUDY

Generally speaking, we do not recommend using point-and-click approaches for data analysis. Instead, we recommend scripting languages, such as R, since they are more flexible and greatly facilitate reproducibility. Similarly, we recommend against the use of point-and-click approaches to organizing files and document preparation. In this part of the book, we demonstrate alternative approaches. Specifically, we will learn to use freely available tools that, although at first may seem cumbersome and non-intuitive, will eventually make you a much more efficient and productive data scientist.

Three general guiding principles that motivate what we learn here are 1) be systematic when organizing your filesystem, 2) automate when possible, and 3) minimize the use of the mouse. As you become more proficient at coding, you will find that 1) you want to minimize the time you spend remembering what you called a file or where you put it, 2) if you find yourself repeating the same task over and over, there is probably a way to automate, and 3) anytime your fingers leave the keyboard, it results in loss of productivity.

A data analysis project is not always a dataset and a script. A typical data analysis challenge may involve several parts, each involving several data files, including files containing the scripts we use to analyze data. Keeping all this organized can be challenging. We will learn to use the *Unix shell* as a tool for managing files and directories on your computer system. Using Unix will permit you to use the keyboard, rather than the mouse, when creating folders, moving from directory to directory, and renaming, deleting, or moving files. We also provide specific suggestions on how to keep the filesystem organized.

The data analysis process is also iterative and adaptive. As a result, we are constantly editing our scripts and reports. In this chapter, we introduce you to the version control system *Git*, which is a powerful tool for keeping track of these changes. We also introduce you to GitHub[6], a service that permits you to host and share your code. We will demonstrate how you can use this service to facilitate collaborations. Another positive benefit of using GitHub is that you can easily showcase your work to potential employers.

Finally, we learn to write reports in R markdown, which permits you to incorporate text and code into a single document. We will demonstrate how, using the `knitr` package, we can write reproducible and aesthetically pleasing reports by running the analysis and generating the report simultaneously.

We will put all this together using the powerful integrated desktop environment RStudio[7]. Throughout the chapter we will be building up an example on US gun murders. The final project, which includes several files and folders, can be seen in the GitHub repository *rairizarry/murders*[8].

[6] http://github.com
[7] https://www.rstudio.com/
[8] https://github.com/rairizarry/murders

ISTUDY

18

Organizing with Unix

Unix is the operating system of choice in data science. We will introduce you to the Unix way of thinking using an example: how to keep a data analysis project organized. We will learn some of the most commonly used commands along the way. However, we won't go into advanced details. We highly encourage you to learn more, especially when you find yourself using the mouse or performing a repetitive task often. In those cases, there is probably a more efficient way to do it in Unix. Here are some basic courses to get you started:

- "Learn the Command Line" through codecademy[1]
- "LinuxFoundationX: Introduction to Linux" through edX[2]
- "The Unix Workbench" through coursera[3]

There are many reference books[4] as well. Bite Size Linux[5] and Bite Size Command Line[6] are two particularly clear, succinct, and complete examples.

When searching for Unix resources, keep in mind that other terms used to describe what we will learn here are *Linux*, *the shell*, and *the command line*. Basically, what we are learning is a series of commands and a way of thinking that facilitates the organization of files without using the mouse.

To serve as motivation, we are going to start constructing a directory using Unix tools and RStudio.

18.1 Naming convention

Before you start organizing projects with Unix you want to pick a name convention that you will use to systematically name your files and directories. This will help you find files and know what is in them.

In general you want to name your files in a way that is related to their contents and specifies how they relate to other files. The Smithsonian Data Management Best Practices[7] has "five precepts of file naming and organization" and they are:

[1] https://www.codecademy.com/learn/learn-the-command-line
[2] https://www.edx.org/course/introduction-linux-linuxfoundationx-lfs101x-1
[3] https://www.coursera.org/learn/unix
[4] https://www.quora.com/Which-are-the-best-Unix-Linux-reference-books
[5] https://gumroad.com/l/bite-size-linux
[6] https://jvns.ca/blog/2018/08/05/new-zine--bite-size-command-line/
[7] https://library.si.edu/sites/default/files/tutorial/pdf/filenamingorganizing20180227.pdf

DOI: 10.1201/9781003220923-18

- Have a distinctive, human-readable name that gives an indication of the content.
- Follow a consistent pattern that is machine-friendly.
- Organize files into directories (when necessary) that follow a consistent pattern.
- Avoid repetition of semantic elements among file and directory names.
- Have a file extension that matches the file format (no changing extensions!)

For specific recommendations we highly recommend you follow The Tidyverse Style Guide[8].

18.2 The terminal

Instead of clicking, dragging, and dropping to organize our files and folders, we will be typing Unix commands into the terminal. The way we do this is similar to how we type commands into the R console, but instead of generating plots and statistical summaries, we will be organizing files on our system.

The terminal is integrated into Mac and Linux systems, but Windows users will have to install an *emulator*. Once you have a terminal open, you can start typing commands. You should see a blinking cursor at the spot where what you type will show up. This position is called the *command line*. Once you type something and hit enter on Windows or return on the Mac, Unix will try to execute this command. If you want to try out an example, type this command:

```
echo "hello world"
```

The command `echo` is similar to `cat` in R. Executing this line should print out `hello world`, then return back to the command line.

Notice that you can't use the mouse to move around in the terminal. You have to use the keyboard. To go back to a command you previously typed, you can use the up arrow.

Note that above we included a chunk of code showing Unix commands in the same way we have previously shown R commands. We will make sure to distinguish when the command is meant for R and when it is meant for Unix.

18.3 The filesystem

We refer to all the files, folders, and programs on your computer as *the filesystem*. Keep in mind that folders and programs are also files, but this is a technicality we rarely think about and ignore in this book. We will focus on files and folders for now and discuss programs, or *executables*, in a later section.

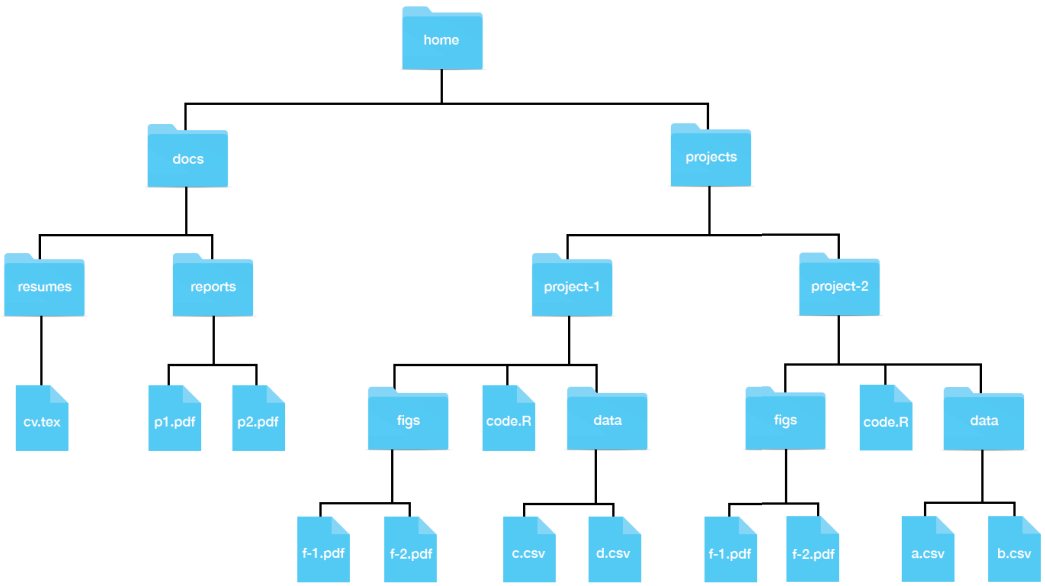
[8] https://style.tidyverse.org/

ISTUDY

18.3.1 Directories and subdirectories

The first concept you need to grasp to become a Unix user is how your filesystem is organized. You should think of it as a series of nested folders, each containing files, folders, and executables.

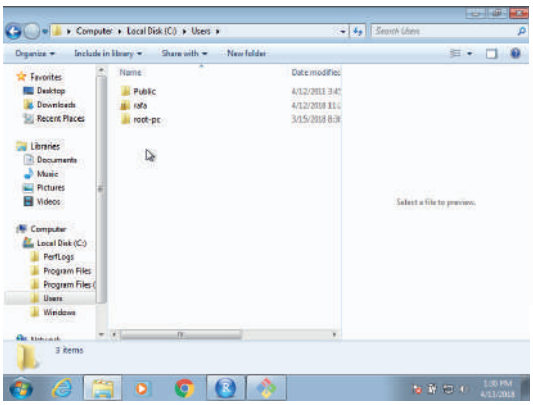
Here is a visual representation of the structure we are describing:

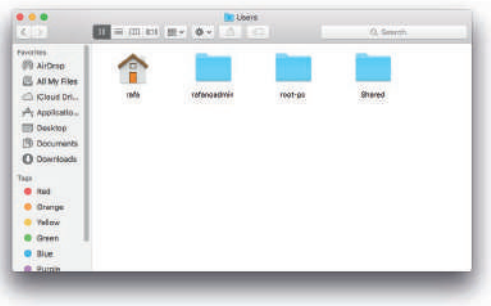


In Unix, we refer to folders as *directories*. Directories that are inside other directories are often referred to as *subdirectories*. So, for example, in the figure above, the directory *docs* has two subdirectories: *reports* and *resumes*, and *docs* is a subdirectory of *home*.

18.3.2 The home directory

The *home* directory is where all your stuff is kept, as opposed to the system files that come with your computer, which are kept elsewhere. In the figure above, the directory called *home* represents your home directory, but that is rarely the name used. On your system, the name of your home directory is likely the same as your username on that system. Below are an example on Windows and Mac showing a home directory, in this case, named *rafa*:

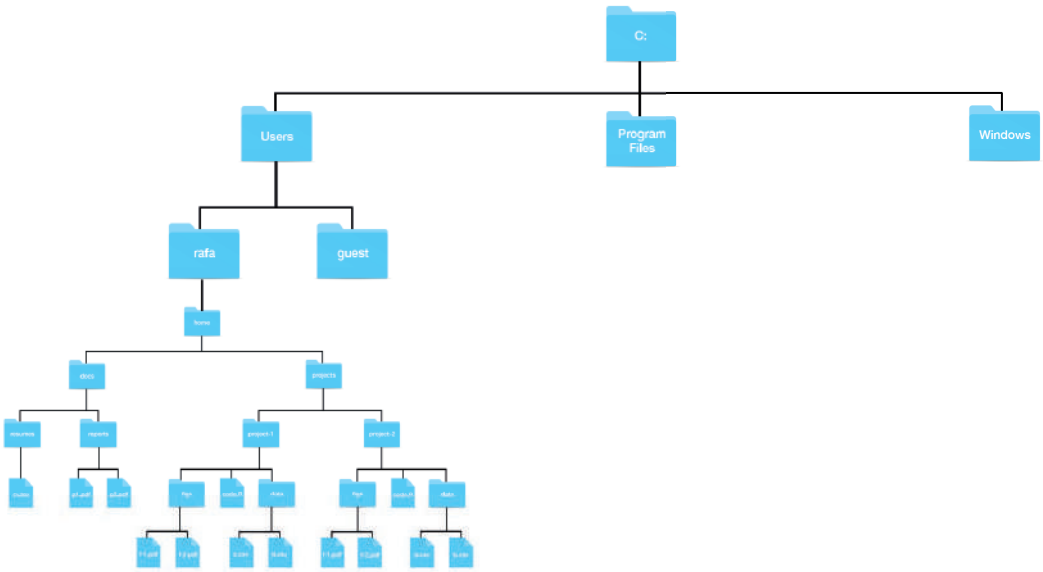




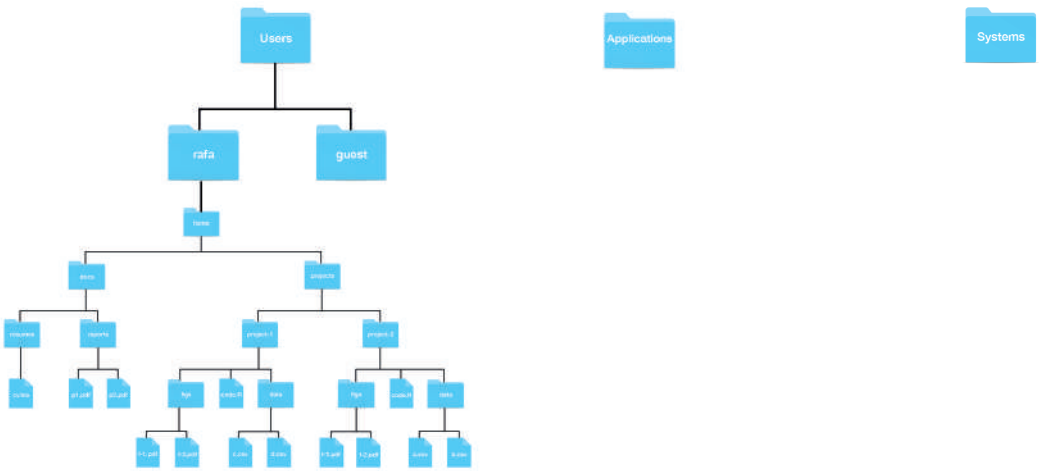
ISTUDY

Now, look back at the figure showing a filesystem. Suppose you are using a point-and-click system and you want to remove the file *cv.tex*. Imagine that on your screen you can see the *home* directory. To erase this file, you would double click on the *home* directory, then *docs*, then *resumes*, and then drag *cv.tex* to the trash. Here you are experiencing the hierarchical nature of the system: *cv.tex* is a file inside the *resumes* directory, which is a subdirectory inside the *docs* directory, which is a subdirectory of the *home* directory.

Now suppose you can't see your home directory on your screen. You would somehow need to make it appear on your screen. One way to do this is to navigate from what is called the *root* directory all the way to your home directory. Any filesystem will have what is called a *root* directory, which is the directory that contains all directories. The *home* directory shown in the figure above will usually be two or more levels from the root. On Windows, you will have a structure like this:



while on the Mac, it will be like this:



ISTUDY

> ⚠ Warning
>
> On Windows, the typical R installation will make your *Documents* directory your home directory in R. This will likely be different from your home directory in Git Bash. Generally, when we discuss home directories, we refer to the Unix home directory which for Windows, in this book, is the Git Bash Unix directory.

18.3.3 Working directory

The concept of a *current location* is part of the point-and-click experience: at any given moment we are *in a folder* and see the content of that folder. As you search for a file, as we did above, you are experiencing the concept of a current location: once you double click on a directory, you change locations and are now *in that folder*, as opposed to the folder you were in before.

In Unix, we don't have the same visual cues, but the concept of a *current location* is indispensable. We refer to this as the *working directory*. Each terminal window you have open has a working directory associated with it.

How do we know what is our working directory? To answer this, we learn our next Unix command: `pwd`, which stands for *print working directory*. This command returns the working directory.

Open a terminal and type:

```
pwd
```

We do not show the result of running this command because it will be quite different on your system compared to others. If you open a terminal and type `pwd` as your first command, you should see something like `/Users/yourusername` on a Mac or something like `/c/Users/yourusername` on Windows. The character string returned by calling `pwd` represents your working directory. When we first open a terminal, it will start in our home directory so in this case the working directory is the home directory.

Notice that the forward slashes `/` in the strings above separate directories. So, for example, the location `/c/Users/rafa` implies that our working directory is called `rafa` and it is a subdirectory of `Users`, which is a subdirectory of `c`, which is a subdirectory of the root directory. The root directory is therefore represented by just a forward slash: `/`.

18.3.4 Paths

We refer to the string returned by `pwd` as the *full path* of the working directory. The name comes from the fact that this string spells out the *path* you need to follow to get to the directory in question from the root directory. Every directory has a full path. Later, we will about *relative paths*, which tell us how to get to a directory from the working directory.

In Unix, we use the shorthand `~` as a nickname for your home directory. So, for example, if `docs` is a directory in your home directory, the full path for *docs* can be written like this `~/docs`.

Most terminals will show the path to your working directory right on the command line. If you are using default settings and open a terminal on the Mac, you will see that right at the command line you have something like `computername:~ username` with `~` representing your working directory, which in this example is the home directory `~`. The same is true for the

ISTUDY

Git Bash terminal where you will see something like `username@computername MINGW64 ~`, with the working directory at the end. When we change directories, we will see this change on both Macs and Windows.

18.4 Unix commands

We will now learn a series of Unix commands that will permit us to prepare a directory for a data science project. We also provide examples of commands that, if you type into your terminal, will return an error. This is because we are assuming the filesystem in the earlier diagram. Your filesystem is different. In the next section, we will provide examples that you can type in.

18.4.1 `ls`: Listing directory content

In a point-and-click system, we know what is in a directory because we see it. In the terminal, we do not see the icons. Instead, we use the command `ls` to list the directory content.

To see the content of your home directory, open a terminal and type:

```
ls
```

We will see more examples soon.

18.4.2 `mkdir` and `rmdir`: make and remove a directory

When we are preparing for a data science project, we will need to create directories. In Unix, we can do this with the command `mkdir`, which stands for *make directory*.

Because you will soon be working on several projects, we highly recommend creating a directory called *projects* in your home directory.

You can try this particular example on your system. Open a terminal and type:

```
mkdir projects
```

If you do this correctly, nothing will happen: no news is good news. If the directory already exists, you will get an error message and the existing directory will remain untouched.

To confirm that you created the directory, you can list the contents of the current working directory:

```
ls
```

You should see the directory we just created listed. Perhaps you can also see many other directories that were already on your computer.

For illustrative purposes, let's make a few more directories. You can list more than one directory name like this:

```
mkdir docs teaching
```

You can check to see if the three directories were created:

```
ls
```

ISTUDY

If you made a mistake and need to remove the directory, you can use the command `rmdir` to remove it.

```
mkdir junk
rmdir junk
```

This will remove the directory as long as it is empty. If it is not empty, you will get an error message and the directory will remain untouched. To remove directories that are not empty, we will learn about the command `rm` later.

18.4.3 `cd`: navigating the filesystem by changing directories

Next we want to create directories inside directories that we have already created. We also want to avoid pointing and clicking our way through the filesystem. We explain how to do this in Unix, using the command line.

Suppose we open a terminal and our working directory is our home directory. We want to change our working directory to `projects`. We do this using the `cd` command, which stands for *change directory*:

```
cd projects
```

To check that the working directory changed, we can use a command we previously learned to see our location:

```
pwd
```

Our working directory should now be `~/projects`. Note that on your computer the home directory `~` will be spelled out to something like `/c/Users/yourusername`.

> **Tip**
>
> In Unix you can auto-complete by hitting tab. This means that we can type `cd d` then hit tab. Unix will either auto-complete if `docs` is the only directory/file starting with `d` or show you the options. Try it out! Using Unix without auto-complete can make it unbearable.

When using `cd`, we can either type a full path, which will start with `/` or `~`, or a *relative path*. In the example above, in which we typed `cd projects`, we used a relative path. If the path you type does not start with `/` or `~`, Unix will assume you are typing a relative path, meaning that it will look for the directory in your current working directory. So something like this will give you an error:

```
cd Users
```

because there is no `Users` directory in your working directory.

Now suppose we want to move back to the directory in which `projects` is a subdirectory, referred to as the *parent directory*. We could use the full path of the parent directory, but Unix provides a shortcut for this: the parent directory of the working directory is represented with two dots: `..` so to move back we simply type:

```
cd ..
```

You should now be back in your home directory which you can confirm using `pwd`.

ISTUDY

Because we can use full paths with `cd`, the following command:

```
cd ~
```

will always take us back to the home directory, no matter where we are in the filesystem.

The working directory also has a nickname, which is a single `.` so if you type

```
cd .
```

you will not move. Although this particular use of `.` is not useful, this nickname does come in handy sometimes. The reasons are not relevant for this section, but you should still be aware of this fact.

In summary, we have learned that when using `cd` we either stay put, move to a new directory using the desired directory name, or move back to the parent directory using `..`

When typing directory names, we can concatenate directories with the forward-slashes. So if we want a command that takes us to the `projects` directory no matter where we are in the filesystem, we can type:

```
cd ~/projects
```

which is equivalent to writing the entire path out. For example, in Windows we would write something like

```
cd /c/Users/yourusername/projects
```

The last two commands are equivalent and in both cases we are typing the full path.

We can also concatenate directory names for relative paths. For instance, if we want to move back to the parent directory of the parent directory of the working directory, we can type:

```
cd ../..
```

Here are a couple of final tips related to the `cd` command. First, you can go back to whatever directory you just left by typing:

```
cd -
```

This can be useful if you type a very long path and then realize you want to go back to where you were, and that too has a very long path.

Second, if you just type:

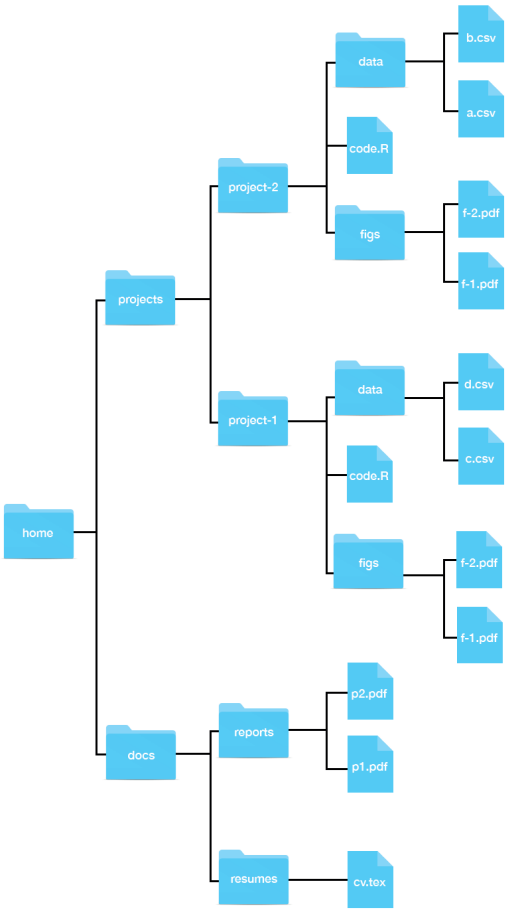
```
cd
```

you will be returned to your home directory.

18.4.4 Examples

Let's explore some examples of using `cd`. To help visualize, we will show the graphical representation of our filesystem vertically:

ISTUDY



Suppose our working directory is `~/projects` and we want to move to `figs` in `project-1`.

Here it is convenient to use relative paths:

```
cd project-1/figs
```

Now suppose our working directory is `~/projects` and we want to move to `reports` in `docs`, how can we do this?

One way is to use relative paths:

```
cd ../docs/reports
```

Another is to use the full path:

```
cd ~/docs/reports
```

If you are trying this out on your system, remember to use auto-complete.

Let's examine one more example. Suppose we are in `~/projects/project-1/figs` and want to change to `~/projects/project-2`. Again, there are two ways.

With relative paths:

```
cd ../../project-2
```

ISTUDY

and with full paths:

```
cd ~/projects/project-2
```

18.5 More Unix commands

18.5.1 `mv`: moving files

In a point-and-click system, we move files from one directory to another by dragging and dropping. In Unix, we use the `mv` command.

> ⚠ Warning
>
> `mv` will not ask "are you sure?" if your move results in overwriting a file.

Now that you know how to use full and relative paths, using `mv` is relatively straightforward. The general form is:

```
mv path-to-file path-to-destination-directory
```

For example, if we want to move the file `cv.tex` from `resumes` to `reports`, you could use the full paths like this:

```
mv ~/docs/resumes/cv.tex ~/docs/reports/
```

You can also use relative paths. You could do this:

```
cd ~/docs/resumes
mv cv.tex ../reports/
```

or this:

```
cd ~/docs/reports/
mv ../resumes/cv.tex ./
```

Notice that in the last one we used the working directory shortcut `.` to give a relative path as the destination directory.

We can also use `mv` to change the name of a file. To do this, instead of the second argument being the destination directory, it also includes a filename. So, for example, to change the name from `cv.tex` to `resume.tex`, we simply type:

```
cd ~/docs/resumes
mv cv.tex resume.tex
```

We can also combine the move and a rename. For example:

```
cd ~/docs/resumes
mv cv.tex ../reports/resume.tex
```

And we can move entire directories. To move the `resumes` directory into `reports`, we do as follows:

```
mv ~/docs/resumes ~/docs/reports/
```

ISTUDY

It is important to add the last `/` to make it clear you do not want to rename the `resumes` directory to `reports`, but rather move it into the `reports` directory.

18.5.2 `cp`: copying files

The command `cp` behaves similar to `mv` except instead of moving, we copy the file, meaning that the original file stays untouched.

So in all the `mv` examples above, you can switch `mv` to `cp` and they will copy instead of move with one exception: we can't copy entire directories without learning about arguments, which we do later.

18.5.3 `rm`: removing files

In point-and-click systems, we remove files by dragging and dropping them into the trash or using a special click on the mouse. In Unix, we use the `rm` command.

> ⚠ Warning
>
> Unlike throwing files into the trash, `rm` is permanent. Be careful!

The general way it works is as follows:

```
rm filename
```

You can actually list files as well like this:

```
rm filename-1 filename-2 filename-3
```

You can use full or relative paths. To remove non-empty directories, you will have to learn about arguments, which we do later.

18.5.4 `less`: looking at a file

Often you want to quickly look at the content of a file. If this file is a text file, the quickest way to do is by using the command `less`. To look a the file `cv.tex`, you do this:

```
cd ~/docs/resumes
less cv.tex
```

To exit the viewer, you type `q`. If the files are long, you can use the arrow keys to move up and down. There are many other keyboard commands you can use within `less` to, for example, search or jump pages.

If you are wondering why the command is called `less`, it is because the original was called `more`, as in "show me more of this file". The second version was called `less` because of the saying "less is more".

ISTUDY

18.6 Case study: Preparing for a project

We are now ready to prepare a directory for a project. We will use the US murders project[9] as an example.

You should start by creating a directory where you will keep all your projects. We recommend a directory called *projects* in your home directory. To do this you would type:

```
cd ~
mkdir projects
```

Our project relates to gun violence murders so we will call the directory for our project `murders`. It will be a subdirectory in our projects directories. In the `murders` directory, we will create two subdirectories to hold the raw data and intermediate data. We will call these `data` and `rda`, respectively.

Open a terminal and make sure you are in the home directory:

```
cd ~
```

Now run the following commands to create the directory structure we want. At the end, we use `ls` and `pwd` to confirm we have generated the correct directories in the correct working directory:

```
cd projects
mkdir murders
cd murders
mkdir data rdas
ls
pwd
```

Note that the full path of our `murders` dataset is `~/projects/murders`.

So if we open a new terminal and want to navigate into that directory we type:

```
cd projects/murders
```

In Section 20.3 we will describe how we can use RStudio to organize a data analysis project, once these directories have been created.

18.7 Advanced Unix

Most Unix implementations include a large number of powerful tools and utilities. We have just learned the very basics here. We recommend that you use Unix as your main file management tool. It will take time to become comfortable with it, but as you struggle, you will find yourself learning just by looking up solutions on the internet. In this section, we superficially cover slightly more advanced topics. The main purpose of the section is to make you aware of what is available rather than explain everything in detail.

[9] https://github.com/rairizarry/murders

ISTUDY

18.7.1 Arguments

Most Unix commands can be run with arguments. Arguments are typically defined by using a dash `-` or two dashes `--` (depending on the command) followed by a letter or a word. An example of an argument is the `-r` behind `rm`. The `r` stands for recursive, and the result is that files and directories are removed recursively, which means that if you type:

```
rm -r directory-name
```

all files, subdirectories, files in subdirectories, subdirectories in subdirectories, and so on, will be removed. This is equivalent to throwing a folder in the trash, except you can't recover it. Once you remove it, it is deleted for good. Often, when you are removing directories, you will encounter files that are protected. In such cases, you can use the argument `-f` which stands for `force`.

You can also combine arguments. For instance, to remove a directory regardless of protected files, you type:

```
rm -rf directory-name
```

Remember that once you remove there is no going back, so use this command very carefully.

A command that is often called with argument is `ls`. Here are some examples:

```
ls -a
```

The `a` stands for all. This argument makes `ls` show you all files in the directory, including hidden files. In Unix, all files starting with a `.` are hidden. Many applications create hidden directories to store important information without getting in the way of your work. An example is `git` (which we cover in depth in Chapter 19). Once you initialize a directory as a git directory with `git init`, a hidden directory called `.git` is created. Another hidden file is the `.gitignore` file.

Another example of using an argument is:

```
ls -l
```

The `l` stands for long and the result is that more information about the files is shown.

It is often useful to see files in chronological order. For that we use:

```
ls -t
```

and to reverse the order of how files are shown you can use:

```
ls -r
```

We can combine all these arguments to show more information for all files in reverse chronological order:

```
ls -lart
```

Each command has a different set of arguments. In the next section, we learn how to find out what they each do.

18.7.2 Getting help

As you may have noticed, Unix uses an extreme version of abbreviations. This makes it very efficient, but hard to guess how to call commands. To make up for this weakness, Unix

ISTUDY

includes complete help files or *man pages* (man is short for manual). In most systems, you can type `man` followed by the command name to get help. So for `ls`, we would type:

```
man ls
```

This command is not available in some of the compact implementations of Unix, such as Git Bash. An alternative way to get help that works on Git Bash is to type the command followed by `--help`. So for `ls`, it would be as follows:

```
ls --help
```

18.7.3 Pipes

The help pages are typically long and if you type the commands above to see the help, it scrolls all the way to the end. It would be useful if we could save the help to a file and then use `less` to see it. The `pipe`, written like this `|`, does something similar. It *pipes* the results of a command to the command after the `pipe`. This is similar to the pipe `|>` that we use in R. To get more help we thus can type:

```
man ls | less
```

or in Git Bash:

```
ls --help | less
```

This is also useful when listing files with many files. We can type:

```
ls -lart | less
```

18.7.4 Wild cards

Some of the most powerful aspects of Unix are the *wild cards*. Suppose we want to remove all the temporary html files produced while trouble shooting for a project. Imagine there are dozens of files. It would be quite painful to remove them one by one. In Unix, we can actually write an expression that means all the files that end in `.html`. To do this we type *wild card*: `*`. As discussed in the data wrangling part of this book, this character means any number of any combination of characters. Specifically, to list all html files, we would type:

```
ls *.html
```

To remove all html files in a directory, we would type:

```
rm *.html
```

The other useful wild card is the `?` symbol. This means any single character. So if all the files we want to erase have the form `file-001.html` with the numbers going from 1 to 999, we can type:

```
rm file-???.html
```

This will only remove files with that format.

We can combine wild cards. For example, to remove all files with the form `file-001` up to `file-999` regardless of suffix, we can type:

```
rm file-???.*
```

ISTUDY

> ⚠ Warning
>
> Combining rm with the `*` wild card can be dangerous. There are combinations of these commands that will erase your entire filesystem without asking "are you sure?". So make sure you understand how it works before using this wild card with the `rm` command. We recommend always checking what files will match the wild card first by using the `ls` command.

18.7.5 Environment variables

Unix has settings that affect your command line *environment*. These are called environment variables. The home directory is one of them. We can actually change some of these. In Unix, variables are distinguished from other entities by adding a `$` in front. The home directory is stored in `$HOME`.

Earlier we saw that `echo` is the Unix command for print. So we can see our home directory by typing:

```
echo $HOME
```

You can see them all by typing:

```
env
```

You can change some of these environment variables. But their names vary across different *shells*. We describe shells in the next section.

18.7.6 Shells

Much of what we use in this chapter is part of what is called the *Unix shell*. There are actually different shells, but the differences are almost unnoticeable. They are also important, although we do not cover those here. You can see what shell you are using by typing:

```
echo $SHELL
```

The most common one is `bash`.

Once you know the shell, you can change environmental variables. In Bash Shell, we do it using `export variable value`. To change the path, described in more detail soon, you would type:

```
export PATH = /usr/bin/
```

> ⚠ Warning
>
> Don't actually run this command though!

There is a program that is run before each terminal starts where you can edit variables so they change whenever you call the terminal. This changes in different implementations, but if using bash, you can create a file called `.bashrc`, `.bash_profile`,`.bash_login`, or `.profile`. You might already have one.

ISTUDY

18.7.7 Executables

In Unix, all programs are files. They are called executables. So `ls`, `mv` and `git` are all files. But where are these program files? You can find out using the command `which`:

```
which git
#> /usr/bin/git
```

That directory is probably full of program files. The directory `/usr/bin` usually holds many program files. If you type:

`ls /usr/bin`

in your terminal, you will see several executable files.

There are other directories that usually hold program files. The Application directory in the Mac or Program Files directory in Windows are examples.

When you type `ls`, Unix knows to run a program which is an executable that is stored in some other directory. So how does Unix know where to find it? This information is included in the environmental variable `$PATH`. If you type:

`echo $PATH`

you will see a list of directories separated by `:`. The directory `/usr/bin` is probably one of the first ones on the list.

Unix looks for program files in those directories in that order. Although we don't teach it here, you can actually create executables yourself. However, if you put it in your working directory and this directory is not on the path, you can't run it just by typing the command. You get around this by typing the full path. So if your command is called my-ls, you can type:

`./my-ls`

Once you have mastered the basics of Unix, you should consider learning to write your own executables as they can help alleviate repetitive work.

18.7.8 Permissions and file types

If you type:

`ls -l`

At the beginning, you will see a series of symbols like this `-rw-r--r--`. This string indicates the type of file: regular file `-`, directory `d`, or executable `x`. This string also indicates the permission of the file: is it readable? writable? executable? Can other users on the system read the file? Can other users on the system edit the file? Can other users execute if the file is executable? This is more advanced than what we cover here, but you can learn much more in a Unix reference book.

18.7.9 Commands you should learn

There are many commands that we do not teach in this book, but we want to make you aware of them and what they do. They are:

ISTUDY

- open/start – On the Mac, `open filename` tries to figure out the right application of the filename and open it with that application. This is a very useful command. On Git Bash, you can try `start filename`. Try opening an `R` or `Rmd` file with `open` or `start`: it should open them with RStudio.
- nano – Opens a bare-bones text editor.
- tar – Archives files and subdirectories of a directory into one file.
- ssh – Connects to another computer.
- find – Finds files by filename on your system.
- grep – Searches for patterns in a file.
- awk/sed – These are two very powerful commands that permit you to find specific strings in files and change them.
- ln – Creates a symbolic link. We do not recommend its use, but you should be familiar with it.

18.7.10 File manipulation in R

We can also perform file management from within R. The key functions to learn about can be seen by looking at the help file for `?files`[10]. Another useful function is `unlink`.

Although not generally recommended, you can run Unix commands in R using `system`.

[10] https://rdrr.io/r/base/files.html

ISTUDY

19

Git and GitHub

Here we provide a brief introduction Git and GitHub. We are only scratching the surface. To learn more about Git, we highly recommend the following resources:

- "Learn Git & GitHub" on Codecademy[1]
- "Hello World" exercises GitHub Guides[2]

If you plan to use Git and Github frequently in conjunction with R, we highly recommend reading Happy Git and GitHub for the useR[3] to learn about the details we don't cover here.

19.1 Why use Git and GitHub?

Three primary reasons to use Git and GitHub are:

1. **Version Control:** Git allows you to track changes in your code, revert to previous file versions, and work on multiple branches simultaneously. Once changes are finalized, different branches can be merged.
2. **Collaboration:** GitHub offers a central storage solution for projects and lets you add collaborators. These collaborators can make changes, keeping all versions synchronized. Moreover, the *pull request* feature on GitHub enables others to suggest modifications to your code, which you can then approve or reject.
3. **Sharing:** Beyond its powerful version control and collaboration tools, Git and GitHub serve as a platform to easily share your code with others.

We primarily emphasize the sharing capabilities here. For a deeper dive into its other functionality, please refer to the provided resources above. One major advantage of hosting code on GitHub is the ease with which you can showcase it to potential employers seeking samples of your work. Given that numerous companies and organizations employ version control systems like Git for project collaboration, they may find it commendable that you possess some knowledge of the tool.

[1] https://www.codecademy.com/learn/learn-git
[2] https://guides.github.com/activities/hello-world/
[3] http://happygitwithr.com/

DOI: 10.1201/9781003220923-19

19.2 Overview of Git

To effectively permit version control and collaboration with Git we need to understand the concept of a *repository*, often simply called a *repo*. A repo is a digital storage space where you can save, edit, and track versions of files for a specific project. Think of it as a project folder combined with a detailed logbook. It holds all the files and directories related to the project and also records of every change made, who made it, and when. This allows multiple people to collaborate on a project without overwriting contributions from others. You can also easily revert to previous versions if needed.

Note that Git permits the creation of different *branches* within a repository. This permits working on files in parallel which is particularly useful for testing ideas that involve big changes before incorporating with a stable version. In this book we provide only examples with just one branch. To learn more about how to define and use multiple branches please consult the resources provided above.

A common practice involves hosting central *main branch* on a GitHub repository that all collaborators can access remotely. The main branch is considered the stable official version. Each collaborator also maintains a *local repository* on their computer, allowing them to edit and test changes before committing them to the main repository.

We're going to explore how Git works by following these step:

1. First, you'll learn how to make changes on your computer in what's called the *working directory*.
2. Once you're happy with your changes, you'll move them to the *staging area*. Think of this as preparing or packing your changes.
3. From there, you'll save these changes to your *local repo*, this is like your personal save point on your computer and will generate a new version of the repository in the log.
4. After saving locally, you'll then send, or *push*, these changes to the main storage space where everyone can see them. In our examples, this main storage space is hosted on GitHub, and Git calls it the *upstream repo*.



Now, to work with this strategy, you'll need an account on GitHub. In the next two sections, we'll guide you on how to set up an account and create repos on GitHub.

ISTUDY

19.3 GitHub accounts

Basic GitHub accounts are free. To create one, go to GitHub[4] where you will see a box in which you can sign up.

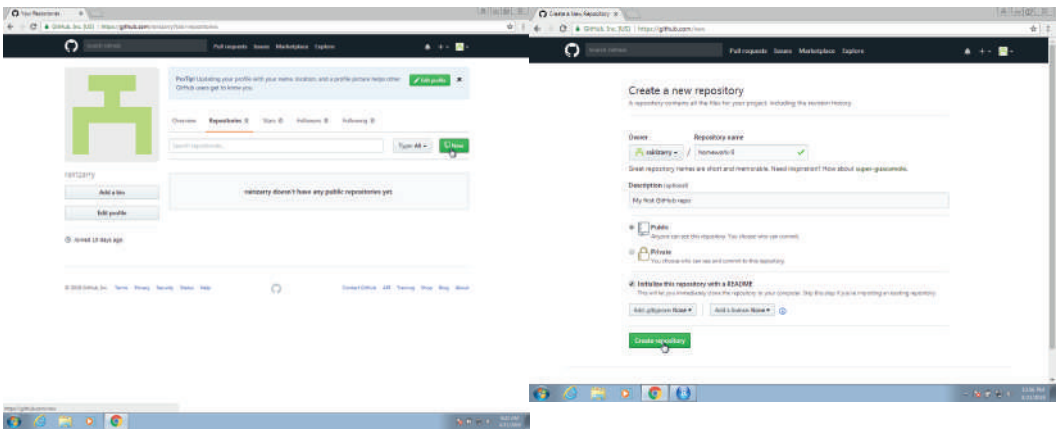
You want to pick a name carefully. It should be short, easy to remember and to spell, somehow related to your name, and professional. This last one is important since you might be sending potential employers a link to your GitHub account. Your initials and last name are usually a good choice.

19.4 GitHub repositories

Once you have an account, you are now ready to create a GitHub repository that will serve as the main or upstream repo for a project. Collaborators you add to this project will be able to manage a local repository on their computer and push changes. Git will help you keep all the different copies synced.

To create a repo, first log in to your account by clicking the *Sign In* button on GitHub. You might already be signed in, in which case the *Sign In* button will not show up. If signing in, you will have to enter your username and password. We recommend you set up your browser to remember this to avoid typing it in each time.

Once on your account, you can click on *Repositories* and then click on *New* to create a new repo. You will be prompted for a name:



When naming your project, pick a descriptive name that clearly tells what the project is about. Keep in mind that as you work on more projects, you'll accumulate many repositories. As an illustration, we will use the name `homework-0`.

You will also be prompted to decide whether your repo should be public or private. To decide, know that this is the difference:

[4] https://github.com/

ISTUDY

- **Public repositories**: Anyone on the internet can see these. Only collaborators can make changes.
- **Private repositories**: Only people you grant access to can view them.

While there are other settings to consider, we typically stick with the default options provided by GitHub.

After creating your repo, GitHub will show you steps to link your local repo (the one on your computer) to the new one you've set up on GitHub. They'll provide some code that you can directly copy and paste into your terminal. We will break down that code so you'll know exactly what each command does.

19.5 Connecting Git and GitHub

When accessing GitHub you need credentials to verify your identity. There are two ways to connect: HTTPS or SSH, each requiring different credentials. We recommend using HTTPS, which uses a Personal Access Token (PAT). Note that **your GitHub website password isn't your access token**.

GitHub provides a detailed guide on obtaining an access token[5] which can be found by searching "Managing your personal access tokens" on the GitHub Docs website[6]. To generate a token:

1. Carefully follow the instructions provided by GitHub.
2. When setting permissions for the token, choose *non-expiring* and select the *repo* option in the *scopes* section.

Once you complete these steps, GitHub will display your token – a lengthy string of characters. You should then:

1. Immediately copy this token to your clipboard. Remember, this is the only time GitHub will show it to you.
2. For security, save this token in a password manager. This ensures you can access it if needed later on.

In some of the procedures outlined below, you'll be prompted to enter your password. Instead, paste the token you've copied. After this, password prompts should no longer appear. If you ever need the token again, retrieve it from your password manager.

For a much more detailed explanation, including how to use SSH instead of HTTPS, please consult Happy Git and GitHub for the useR[7].

[5] https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens
[6] https://docs.github.com/en
[7] https://happygitwithr.com/https-pat

ISTUDY

The next step is to let Git know who we are. This will make it easier to connect with GitHub. To to this type the following two commands in our terminal window:

```
git config --global user.name "Your Name"
git config --global user.mail "your@email.com"
```

This will change the Git configuration in way that anytime you use Git, it will know this information. Note that **you need to use the email account that you used to open your GitHub account**.

19.6 Initial setup

In a terminal, move to the directory you want to store the local repository. We recommend naming the directory the same as the GitHub repo. In our example we would use:

```
mkdir homework-0
cd homework-0
```

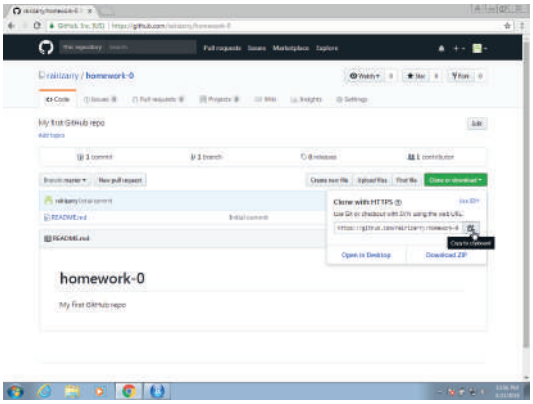
We then initialize the directory as a Git repository, starting the version control process.

```
git init
```

> **i** main verus master
>
> GitHub now uses `main` as the default branch name. In the past, both Git and GitHub used `master` as the default. As a result, many older repositories or older versions of Git might still use `master` as their primary branch.
> To ensure your local branch aligns with the GitHub repository's branch name:
>
> 1. Visit the GitHub repository page.
> 2. Check the dropdown menu on the left that lists branches. This will display the default branch name.
>
> To verify your local branch name, use:
> `git branch`
> If you see a branch name other than `main` but want it to be `main`, rename it with:
> `git branch -M main*`
> The `-M` stands for move. Note that this is different from changing branches, it is renaming the current branch.

To link your local repository to its counterpart on GitHub, you'll need the GitHub repository's URL. To find this, go to the repository's webpage. Click the green *Code* button to quickly copy the URL, which in our example is `https://github.com/rairizarry/homework-0.git`.

ISTUDY



Once you have this you can type

```
git remote add origin https://github.com/rairizarry/homework-0.git
```

To understand this command note that `git remote add` adds a new remote reference. A *remote* in Git refers to another place where your code repository is stored, usually on the internet or another network. `origin` is the conventional name given to the remote repository or the central repository that other people will treat as an main project source. It's essentially a shorthand alias for the repository's URL. You could technically name it anything you want, but `origin` is the convention most use. Finally, `https://github.com/rairizarry/homework-0.git` is the URL of the remote repository. It tells Git where the repository is hosted. Together, these commands set up a new local Git repository and link it to a remote repository on GitHub.

19.7 Git basics

Now that you have initialized a directory to store your local repository, we can learn how to move files from our *working directory* all the way to the upstream repo.

19.7.1 The working directory



The working directory is the same as your Unix working directory. In our example, if we create a file in the `homework-0` directory, it is considered to be in the working directory. Git can tell you how the files in the working directory relate to the versions of the files in other areas with the command

```
git status
```

ISTUDY

Because we have not done anything yet, you should receive a message such as

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

If we add a file, say `code.R`, you will see a message like:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    code.R
```

19.7.2 add



Now we are going to make changes to these files. Eventually, we want these new versions of the files to be tracked and synced with the upstream repo. But we don't want to keep track of every little change: we don't want to sync until we are sure these versions are final enough to share as a new version. For this reason, edits in the staging area are not kept by the version control system.

To demonstrate, we add `code.R` to the staging area:

```
git add code.R
```

Running `git status` now shows

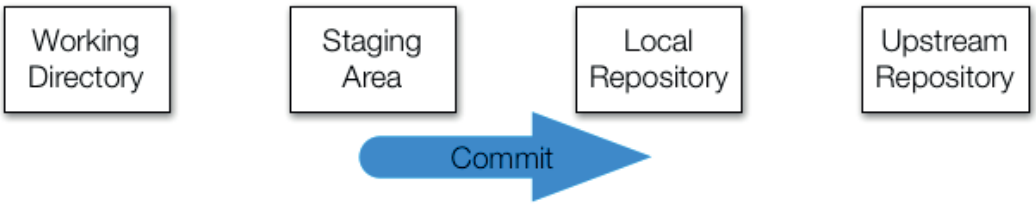
```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   code.R
```

Note that it is not a problem to have other files in the working directory that are not in the staging area. For example, if we create files `test-1.R` and `test-2.R`, `git status` reminds us these are not staged:

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   code.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    test-1.R
    test-2.R
```

ISTUDY

19.7.3 commit



If we are now ready to make a first version of our repository, which only includes `code.R`, we can use the following command:

```
git commit -m "Adding a new file."
```

Note that `commit` requires us to add a message. Making these informative will help us remember why this change was made. After running `commit` we will receive a message letting us know it was committed:

```
[main (root-commit) 1735c25] adding a new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 code.R
```

Note that if we edit `code.R`, it changes only in the working directory. `git status` shows us

```
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   code.R
```

To add the edited file to the local repo, we need to stage the edited file and commit the changes

```
git add code.R
git commit -m "Added some lines of code."
```

which gives us a message letting us know a change was made:

```
[main 8843673] added some lines of code
 1 file changed, 1 insertion(+)
```

Note that we can achieve the same results with just one line by following the commit command with the files we want committed:

```
git commit -m "Added some lines of code." code.R
```

This is convenient when the number of files that change is small and we can list them at the end.

To see version control in action note what happens when we type

```
git log code.R
```

ISTUDY

we get a list of the version that have been stored in our log:

```
commit 88436739dcbd57d8ad27a23663d30fd2c06034ca (HEAD -> main)
Author: Rafael A Irizarry
Date:   Sun Sep 3 15:32:03 2023 -0400

    Added some lines of code.

commit 1735c25c675d23790df1f9cdb3a215a13c8ae5d6
Author: Rafael A Irizarry
Date:   Sun Sep 3 15:27:19 2023 -0400

    Adding a new file.
```

19.7.4 push



Once we are ready to sync our local repo with the upstream GitHub repo, we can use

```
git push -u origin main
```

> **Note**
>
> If this is your first time pushing to your GitHub account, you will be asked for a password and you have to enter the personal access token we described in Section 19.5. You should only need to do this once.

The `-u` flag, short for `--set-upstream` will make Git remember that in this repository you want to push to the `main` branch in the remote repo `origin` defined in the initialization. This is beneficial because the next time you want to push or pull from this branch, you can simply use

```
git push
```

If you need a reminder of where you are pushing to you can type

```
git remove -v
```

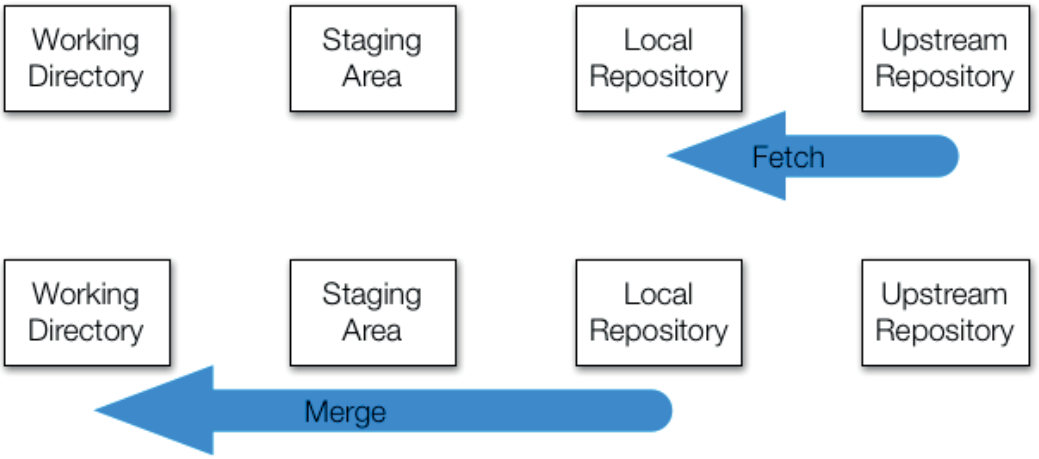
The `v` stands for `verbose`. In our example we will get

```
origin  https://github.com/username/homework-0.git (fetch)
origin  https://github.com/username/homework-0.git (push)
```

We describe `fetch` next. If you don't get anything back it means you have not defined your remote as we did in Section 19.6.

ISTUDY

19.7.5 fetch and merge



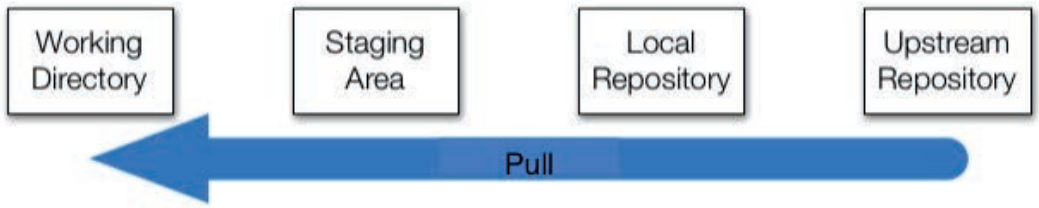
If this is a collaborative project, the upstream repo may change and become different than your version. To update your local repository to be like the upstream repo, we use the command `fetch`:

```
git fetch
```

And then to make these copies to our working directory, we use the command:

```
git merge
```

19.7.6 pull



We very often just want to fetch and merge without checking. For this, we use:

```
git pull
```

19.7.7 clone

You can easily download all the code and version control logs from an existing public repository using `git clone`. When you clone you are essentially making a complete copy of the entire directory. For example, you can download all the code used to create this book using:

```
git clone https://github.com/rafalab/dsbook-part-1.git
```

ISTUDY

You can see a simple example with the murders directory created in the Unix chapter by cloning this repository:

```
git clone https://github.com/rairizarry/murders.git
```

If you use `git clone`, you do not need to initialize as the branch and remote will already be defined. Now, to push changes you need to be added as a collaborator. Otherwise you will have to follow the more complex process of a *pull request*, which we don't cover here.

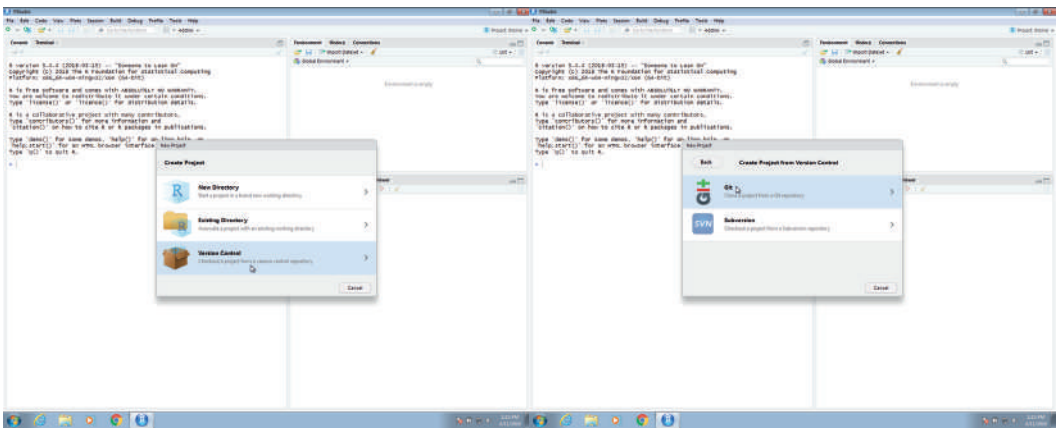
19.8 .gitignore

When we use `git status` we obtain information about all files in our local repo. But we don't necessarily need to add all the files in our working directory to the Git repo, only the ones we want to keep track of or the ones we want to share. If our work is producing files of a certain type that we do not want to keep track of, we can add the suffix that defines these files to the .gitignore file. More details on using .gitignore are included on the git-scm website[8]. These files will stop appearing when you type `git status`.

19.9 Git in RStudio

While command line Git is a powerful and flexible tool, it can be somewhat daunting when we are getting started. RStudio provides a graphical interface that facilitates the use of Git in the context of a data analysis project.

To do this, we start a project but, instead of *New Directory*, or *Existing Directory*, we select *Version Control* and then we will select *Git* as our version control system:

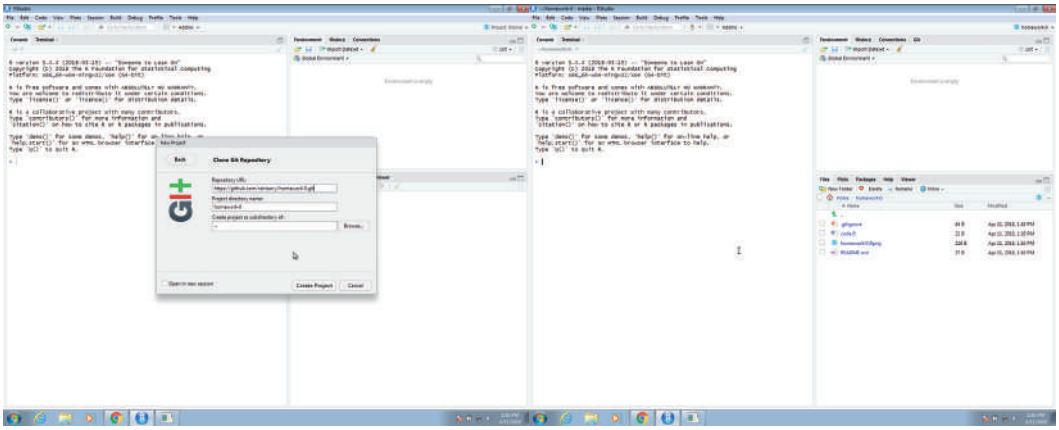


The repository URL is the link you used as `origin` or to clone. In Section 19.4, we used `https://github.com/username/homework-0.git` as an example. In the project directory name, you need to put the name of the folder that was generated, which in our example

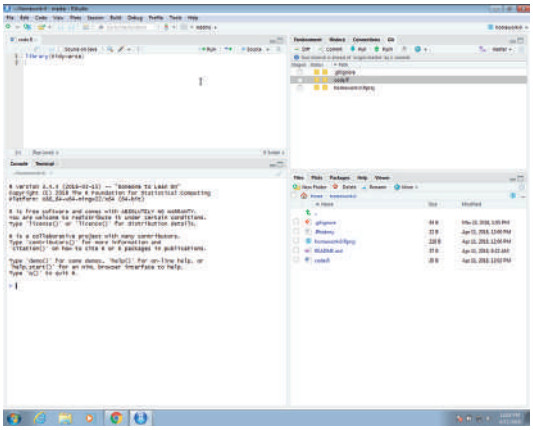
[8] https://git-scm.com/docs/gitignore

ISTUDY

will be the name of the repo `homework-0`. This will create a folder called `homework-0` on your local system. Note you will need to remove the folder if it already exists or chose a different name. Once you do this, the project is created and it is aware of the connection to a GitHub repo. You will see on the top right corner the name and type of project as well as a new tab on the upper right pane titled *Git*.



If you select this tab, it will show you the files in your project, excluding those in `.gitignore`, with some icons that give you information about these files and their relationship to the repo. In the example below, we already added a file to the folder, called *code.R* which you can see in the editing pane.

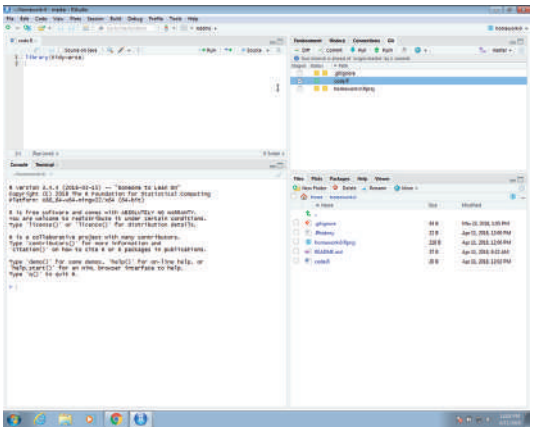


We now need to pay attention to the Git pane. It is important to know that **your local files and the GitHub repo will not be synced automatically**. As described in Section 19.2, you have to sync using git push when you are ready. We show how you can do this through RStudio rather than the terminal below.

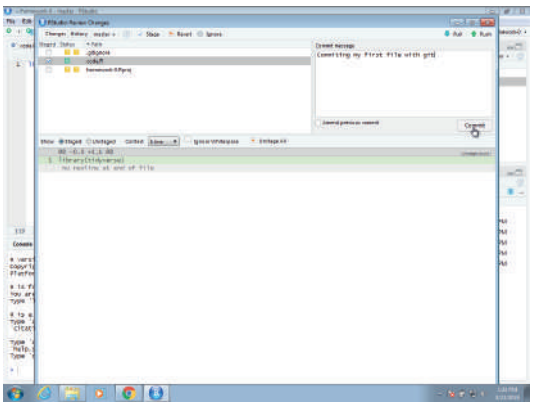
Before we start working on a collaborative project, usually the first thing we do is *pull* in the changes from the remote repo, in our case the one on GitHub. However, for the example shown here, since we are starting with an empty repo and we are the only ones making changes, we don't need to start by pulling.

ISTUDY

In RStudio, the status of the file as it relates to the remote and local repos are represented in the status symbols with colors. A yellow square means that Git knows nothing about this file. To sync with the GitHub repo, we need to *add* the file, then *commit* the change to our local Git repo, then *push* the change to the GitHub repo. Right now, the file is just on our computer. To add the file using RStudio, we click the *Stage* box. You will see that the status icon now changes to a green A.



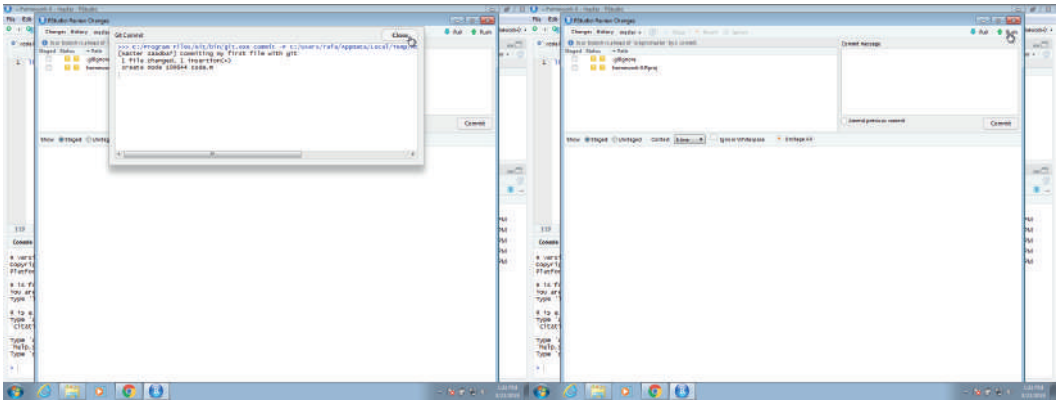
Now we are ready to commit the file to our local repo. In RStudio, we can use the *Commit* button. This will open a new dialog window. With Git, whenever we commit a change, we are required to enter a message describing the changes being *committed*.



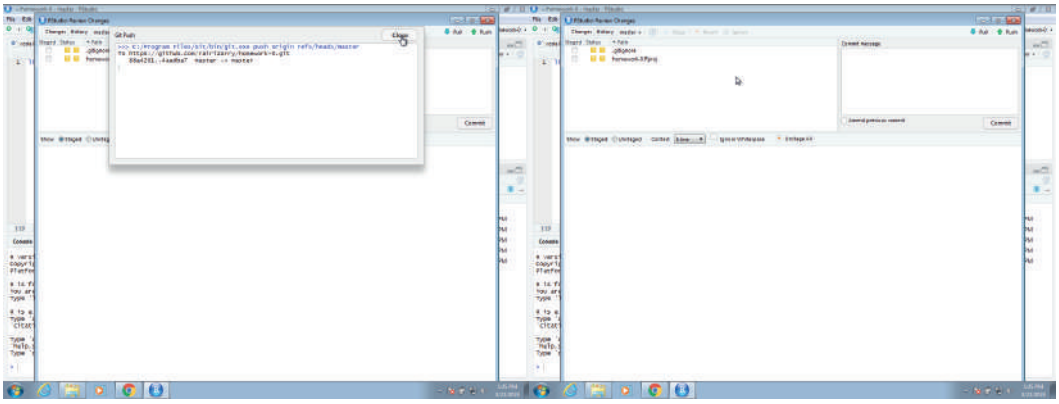
In this case, we will simply describe that we are adding a new script. In this dialog box, RStudio also gives you a summary of what you are changing to the GitHub repo. In this case, because it is a new file, the entire file is highlighted as green, which highlights the changes.

Once we hit the commit button, we should see a message from Git with a summary of the changes that were committed. Now we are ready to *push* these changes to the GitHub repo. We can do this by clicking on the *Push* button on the top right corner:

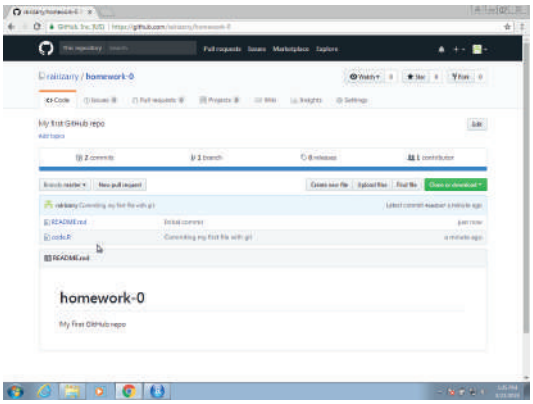
ISTUDY



We now see a message from Git letting us know that the push has succeeded. In the pop-up window we no longer see the `code.R` file. This is because no new changes have been performed since we last pushed. We can exit this pop-up window now and continue working on our code.



If we now visit our repo on the web, we will see that it matches our local copy.



ISTUDY

Congratulations, you have successfully shared code on a GitHub repository!

> **Note**
>
> For the example shown here, we only added *code.R*. But, in general, for an RStudio project, we recommend adding a README.md file and both the .gitignore and .Rproj files.

ISTUDY

20

Reproducible projects

The final product of a data analysis project is often a report. Many scientific publications can be thought of as a final report of a data analysis. The same is true for news articles based on data, an analysis report for your company, or lecture notes for a class on how to analyze data. The reports are often on paper or in a PDF that includes a textual description of the findings along with some figures and tables resulting from the analysis.

Imagine that after you finish the analysis and the report, you are told that you were given the wrong dataset, you are sent a new one and you are asked to run the same analysis. Or what if you realize that a mistake was made and you need to re-examine the code, fix the error, and re-run the analysis? Or imagine that someone you are training wants to see your code and be able to reproduce the results to learn about your approach?

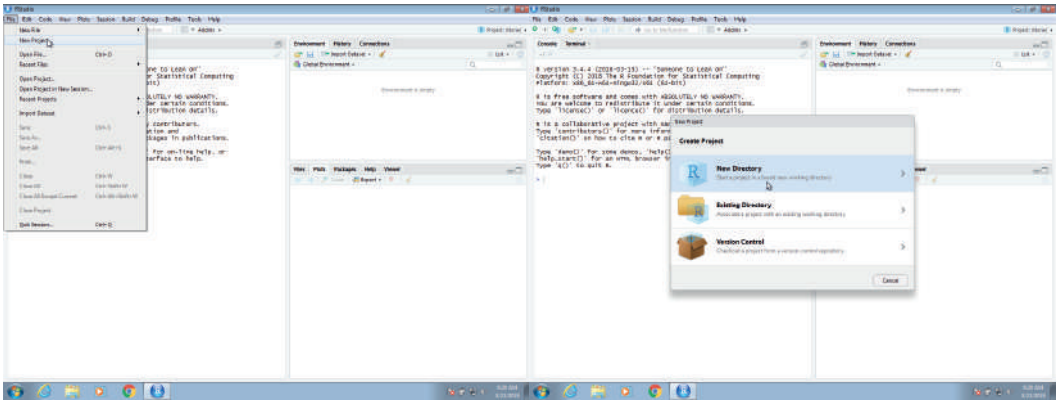
Situations like the ones just described are actually quite common for a data analyst. Here, we describe how you can keep your projects organized with RStudio so that re-running an analysis is straightforward. We then demonstrate how to generate reproducible reports with quarto or R markdown. The **knitR** package will greatly help with recreating reports with minimal work. This is possible due to the fact that markdown documents permit code and textual descriptions to be combined into the same document, and the figures and tables produced by the code are automatically added to the document.

20.1 RStudio projects

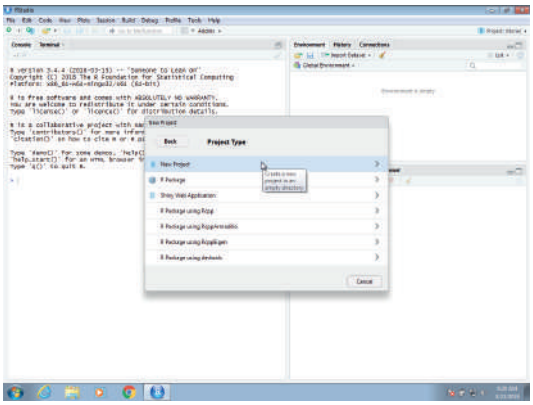
RStudio provides a way to keep all the components of a data analysis project organized into one folder and to keep track of information about this project, such as the Git status of files, in one place. In Section 19.9 we demonstrated how RStudio facilitates the use of Git and GitHub through RStudio projects. In this section we quickly demonstrate how to start a new a project and some recommendations on how to keep these organized. RStudio projects also permit you to have several RStudio sessions open and keep track of which is which.

To start a project, click on *File* and then *New Project*. Often we have already created a folder to save the work, as we did in Section 18.6, and we select *Existing Directory*. Here we show an example in which we have not yet created a folder and select the *New Directory* option.

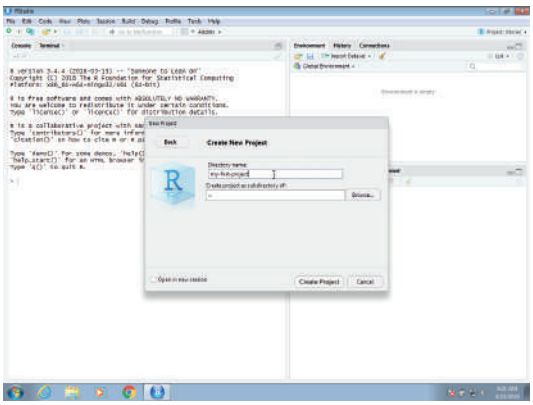
DOI: 10.1201/9781003220923-20



Then, for a data analysis project, you usually select the *New Project* option:

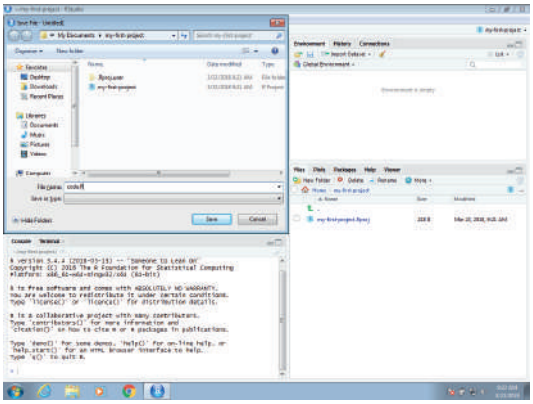


Now you will have to decide on the location of the folder that will be associated with your project, as well as the name of the folder. When choosing a folder name, just like with file names, make sure it is a meaningful name that will help you remember what the project is about. As with files, we recommend using lower case letters, no spaces, and hyphens to separate words. We will call the folder for this project `my-first-project`. This will then generate a *Rproj* file called `my-first-project.Rproj` in the folder associated with the project. We will see how this is useful a few lines below.



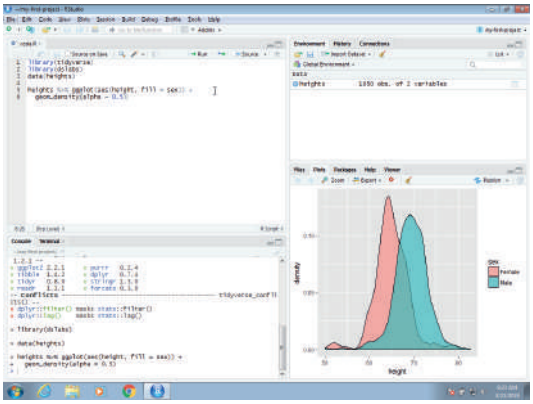
ISTUDY

You will be given options on where this folder should be on your filesystem. In this example, we will place it in our home folder, but this is generally not good practice. As we described in Section 18.6 in the Unix chapter, you want to organize your filesystem following a hierarchical approach and with a folder called *projects* where you keep a folder for each project.

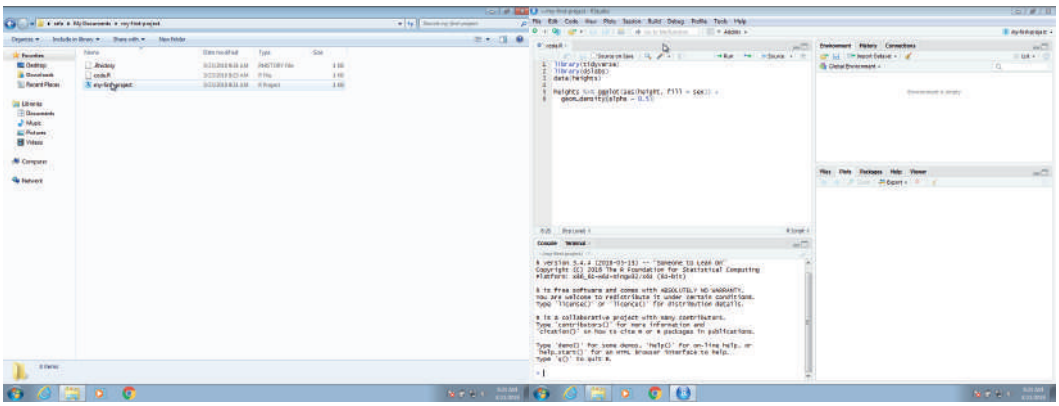


When you start using RStudio with a project, you will see the project name in the upper right corner. This will remind you what project this particular RStudio session belongs to. When you open an RStudio session with no project, it will say *Project: (None)*.

When working on a project, all files will be saved and searched for in the folder associated with the project. Below, we show an example of a script that we wrote and saved with the name `code.R`. Because we used a meaningful name for the project, we can be a bit less informative when we name the files. Although we do not do it here, you can have several scripts open at once. You simply need to click *File*, then *New File* and pick the type of file you want to edit.



One of the main advantages of using Projects is that after closing RStudio, if we wish to continue where we left off on the project, we simply double click or open the file saved when we first created the RStudio project. In this case, the file is called `my-first-project.Rproj`. If we open this file, RStudio will start up and open the scripts we were editing.



Another advantage is that if you click on two or more different Rproj files, you start new RStudio and R sessions for each.

20.2 Markdown

Markdown is a format for *literate programming* documents that is widely used to generate html pages or pdf documents. Literate programming weaves instructions, documentation, and detailed comments in between machine executable code, producing a document that describes the program that is best for human understanding[1]. You can learn more about markdown with online tutorials[2].

Unlike a word processor, such as Microsoft Word, where what you see is what you get, with markdown, you need to *compile* the document into the final report. The markdown document looks different than the final product. This approach seems like a disadvantage at first, but it can save you time in the long run. For example, instead of producing plots and inserting them one by one into the word processing document, the plots are automatically added when the document is compiled. If you need to change the plots, you just recompile the document after editing the code that produces the plot.

In R, we can produce literate programming documents using Quarto or R markdown. We recommend using Quarto because it is a newer and more flexible version of R markdown that permits the use of languages other than R. Because R markdown preceded Quarto by several years, many public and educational literate programming documents are written in R markdown. However, because the format is similar and both use the **knitr** package to execute the R code (details in Section 20.2.4), most existing R markdown files can be rendered with Quarto without modification.

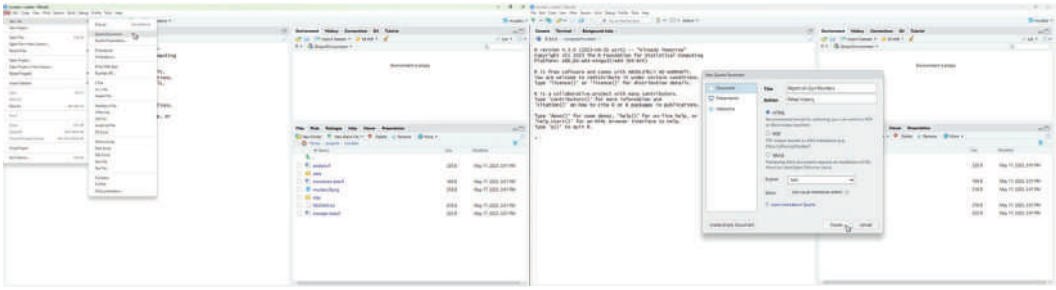
In RStudio, you can start either a Quarto or R markdown document by clicking on *File*, *New File*, then *Quarto Document* or *R Markdown*, respectively. You will then be asked to enter a title and author for your document. We are going to prepare a report on gun murders so we will give it an appropriate name. You can also decide what format you would like the final report to be in: HTML, PDF, or Microsoft Word. Later, we can easily change this, but

[1] Knuth, Donald Ervin. "Literate programming." The Computer Journal 27.2 (1984): 97–111. https://academic.oup.com/comjnl/article/27/2/97/343244

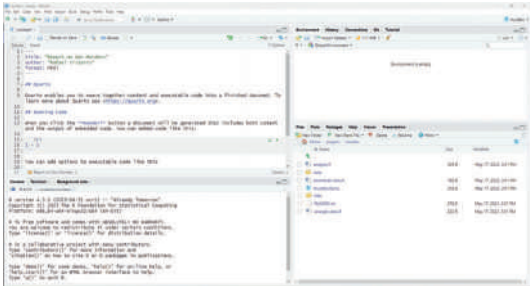
[2] https://www.markdowntutorial.com/

ISTUDY

here we select html as it is the preferred format for debugging purposes:



This will generate a template file:



As a convention, we use `qmd` and `Rmd` suffixes for Quarto and R markdown files, respectively.

Once you gain experience with markdown, you will be able to do this without the template and can simply start from a blank template.

In the template, you will see several things to note.

20.2.1 The header

At the top you see:

```
---
title: "Report on Gun Murders"
author: "Rafael Irizarry"
format: html
---
```

The things between the `---` is the *YAML* header. YAML is a widely used language mainly used for providing configuration data. With Quarto and R markdown it is mainly used to define options for the document. You can define many other things in the header than what is included in the template. We don't discuss those here, but much information is available from the quarto guide[3]. The one parameter that we will highlight is `format`. By changing this to, say, `pdf`, we can control the type of output that is produced when we compile. The title and author parameters are automatically filled because we filled in the blanks in the RStudio dialog box that pops up when creating a new document.

[3] https://quarto.org/docs/guide/

ISTUDY

20.2.2 R code chunks

In various places in the document, we see something like this:

````
```{r}
1 + 1
```
````

These are the code chunks. When you compile the document, the R code inside the chunk, in this case `1+1`, will be evaluated and the result included in that position in the final document.

To add your own R chunks, you can type the characters above quickly with the key binding command-option-I on the Mac and Ctrl-Alt-I on Windows.

This applies to plots as well; the plot will be placed in that position. We can write something like this:

````
```{r}
plot(1)
```
````

By default, the code will show up as well. To avoid having the code show up, you can use an argument, which are annotated with `#|` To avoid showing code in the final document, you can use the argument `echo: FALSE`. For example:

````
```{r}
#| echo: false

1+1
```
````

We recommend getting into the habit of adding a label to the R code chunks. This will be very useful when debugging, among other situations. You do this by adding a descriptive word like this:

````
```{r}
#| label: one-plus-one

1+1
```
````

20.2.3 Global execution options

If you want to apply an option globally, you can include in the header, under `execute`. For example adding the following line to the header make code not show up, by default:

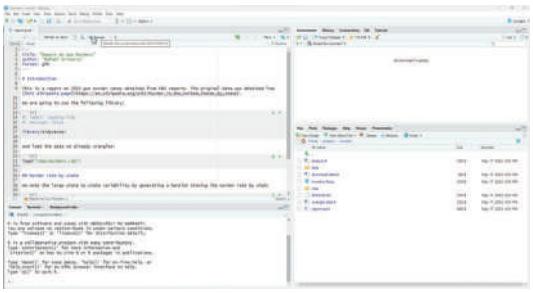
```
execute:
  echo: false
```

We will not cover more details here, but as you become more experienced with R Markdown, you will learn the advantages of setting global options for the compilation process.

ISTUDY

20.2.4 knitR

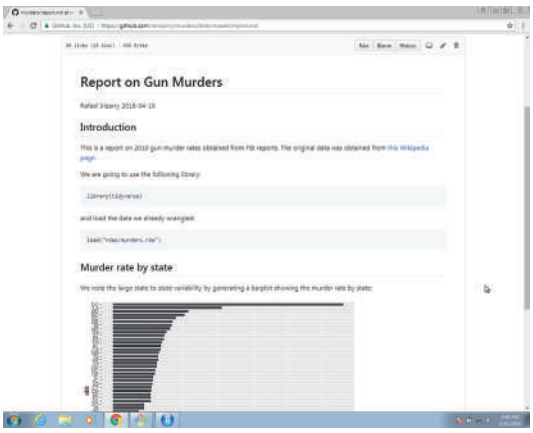
We use the **knitR** package to compile Quarto or R markdown documents. The specific function used to compile is the `knit` function, which takes a filename as input. RStudio provides a button that makes it easier to compile the document. For the screenshot below, we have edited the document so that a report on gun murders is produced. You can see the file on GitHub[4]. You can now click on the `Render` button:



Note that the first time you click on the *Render* button, a dialog box may appear asking you to install packages you need. Once you have installed the packages, clicking *Render* will compile your Quarto file and the resulting document will pop up.

This particular example produces an html document which you can see in your working directory. To view it, open a terminal and list the files. You can open the file in a browser and use this to present your analysis. You can also produce a PDF or Microsoft document by changing: `format: html` to `format: pdf` or `format: docx`. Note this is one difference between Quarto and R markdown. With R markdown we use `output: html_document`, `output: pdf_document`, or `output: word_document`.

We can also produce documents that render on GitHub using `format: gfm`, which stands for GitHub flavored markdown. This will produce a markdown file, with suffix `md`, that renders nicely on GitHub. Because we have uploaded these files to GitHub, you can click on the `md` file and you will see the report as a webpage:



This is a convenient way to share your reports.

[4] https://raw.githubusercontent.com/rairizarry/murders/master/report.qmd

ISTUDY

20.2.5 Learning more

There is a lot more you can do with R markdown. We highly recommend you continue learning as you gain more experience writing reports in R. There are many free resources on the internet including:

- The Quarto Guide[5]
- Hello, Quarto tutorial[6]
- Dynamic Documents with R and knitr textbook[7]

20.3 Organizing a data science project

In this section we put it all together to create the US murders project and share it on GitHub.

20.3.1 Create directories in Unix

In Section 18.6 we demonstrated how to use Unix to prepare for a data science project using an example. Here we continue this example and show how to use RStudio. In Section 18.6 we created the following directories using Unix:

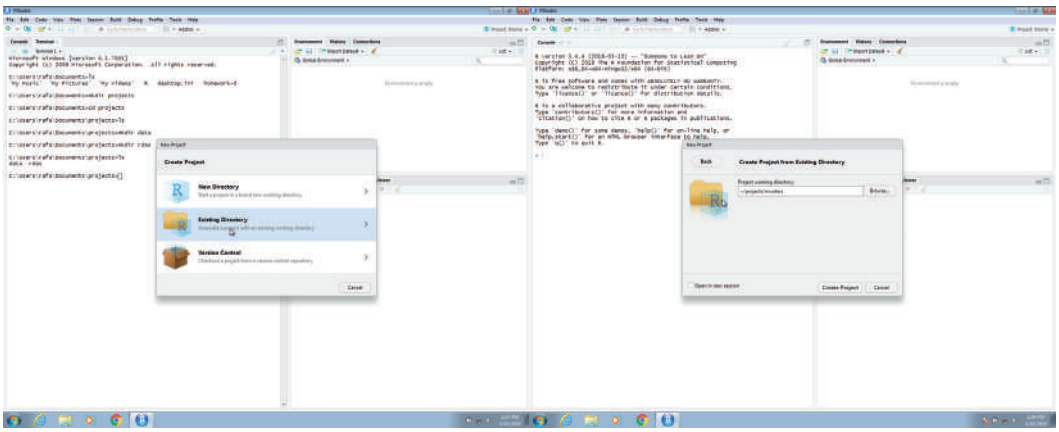
```
cd ~
cd projects
mkdir murders
cd murders
mkdir data rdas
```

20.3.2 Create an RStudio project

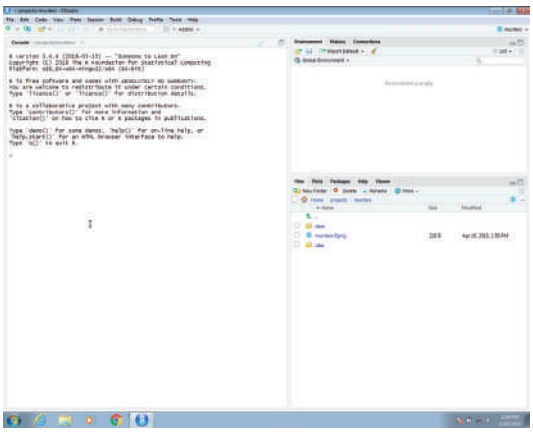
In the next section we will use create an RStudio project. In RStudio we go to *File* and then *New Project...* and when given the options we pick *Existing Directory*. We then write the full path of the `murders` directory created above.

[5] https://quarto.org/docs/guide/
[6] https://quarto.org/docs/get-started/hello/rstudio.html
[7] https://duhi23.github.io/Analisis-de-datos/Yihue.pdf

ISTUDY



Once you do this, you will see the `rdas` and `data` directories you created in the RStudio *Files* tab.



Keep in mind that when we are in this project, our default working directory will be `~/projects/murders`. You can confirm this by typing `getwd()` into your R session. This is important because it will help us organize the code when we need to write file paths.

> **Tip**
>
> Try to always use relative paths in code for data analysis projects. These should be relative to the default working directory. The problem with using full paths, including using the home directory `~` as part of your path, is that your code is unlikely to work on file systems other than yours since the directory structures will be different.

20.3.3 Editting R scripts

Let's now write a script that downloads a file into the data directory. We will call this file `download-data.R`.

ISTUDY

The content of this file will be:

```
url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/
extdata/murders.csv"
dest_file <- "data/murders.csv"
download.file(url, destfile = dest_file)
```

Notice that we are using the relative path `data/murders.csv`.

Run this code in R and you will see that a file is added to the `data` directory.

Now we are ready to write a script to read this data and prepare a table that we can use for analysis. Call the file `wrangle-data.R`. The content of this file will be:

```
library(tidyverse)
murders <- read_csv("data/murders.csv")
murders <- murders |> mutate(region = factor(region),
                             rate = total / population * 10^5)
save(murders, file = "rdas/murders.rda")
```

Again note that we use relative paths exclusively.

20.3.4 Saving processed data

In this file, we introduce an R command we have not seen: `save`. The `save` command in R saves objects into what is called an *rda file*: *rda* is short for R data. We recommend using the `.rda` suffix on files saving R objects. You will see that `.RData` is also used.

If you run this code above, the processed data object will be saved in a file in the `rda` directory. You can then restore the object using `load`. Although not the case here, this approach is often practical because generating the data object we use for final analyses and plots can be a complex and time-consuming process. So we run this process once and save the file. But we still want to be able to generate the entire analysis from the raw data.

While `save` let's you save several objects that then get loaded with the same names used when saving, the function `saveRDS` lets you save one object, without the name. To bring it back you use the `readRDS` function. An example of how this is used is you save it in one session:

```
saveRDS(murders, file = "rdas/murders.rda")
```

Then read it in in another session, using whatever object name you want:

```
dat <- readRDS("rdas/murders.rda")
```

20.3.5 The main analysis file

Now we are ready to write the analysis file. Let's call it `analysis.R`. The content should be the following:

ISTUDY

```
library(tidyverse)
load("rdas/murders.rda")

murders |> mutate(abb = reorder(abb, rate)) |>
  ggplot(aes(abb, rate)) +
  geom_bar(width = 0.5, stat = "identity", color = "black") +
  coord_flip()
```

If you run this analysis, you will see that it generates a plot.

20.3.6 Other directories

Now suppose we want to save the generated plot for use in a report or presentation. We can do this with the **ggplot** command `ggsave`. But where do we put the graph? We should be systematically organized so we will save plots to a directory called `figs`. Start by creating a directory by typing the following in the terminal:

```
mkdir figs
```

and then you can add the line:

```
ggsave("figs/barplot.png")
```

to your R script. If you run the script now, a png file will be saved into the `figs` directory. If we wanted to copy that file to some other directory where we are developing a presentation, we can avoid using the mouse by using the `cp` command in our terminal.

20.3.7 The README file

You now have a self-contained analysis in one directory. One final recommendation is to create a `README.txt` file describing what each of these files does for the benefit of others reading your code, including your future self. This would not be a script but just some notes. One of the options provided when opening a new file in RStudio is a text file. You can save something like this into the text file:

```
We analyze US gun murder data collected by the FBI.

download-data.R - Downloads csv file to data directory

wrangle-data.R - Creates a derived dataset and saves as R object in rdas
directory

analysis.R - A plot is generated and saved in the figs directory.
```

20.3.8 Initializing a Git directory

In Section 19.6 we demonstrated how to initialize a Git directory and connect it to the upstream repository on GitHub, which we already created in that section.

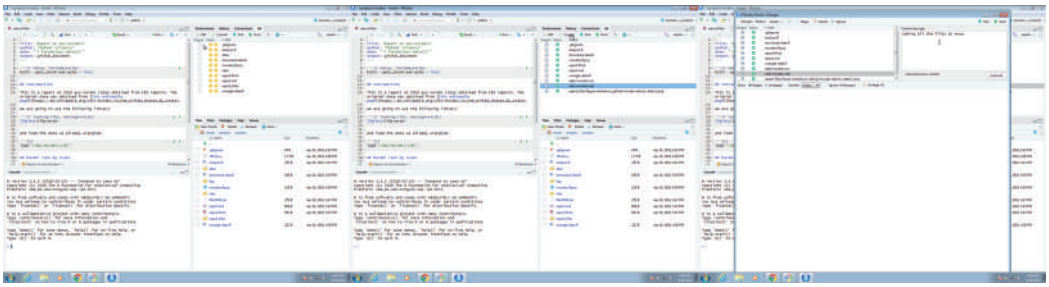
ISTUDY

We can do this in the Unix terminal:

```
cd ~/projects/murders
git init
git add README.txt
git commit -m "First commit. Adding README.txt file just to get started"
git remote add origin `https://github.com/rairizarry/murders.git`
git push -u origin remote
```

20.3.9 Add, commit, and push files using RStudio

We can continue adding and committing each file, but it might be easier to use RStudio. To do this, start the project by opening the Rproj file. The git icons should appear and you can add, commit and push using these.



We can now go to GitHub and confirm that our files are there. You can see a version of this project, organized with Unix directories, on GitHub[8]. You can download a copy to your computer by using the `git clone` command on your terminal. This command will create a directory called `murders` in your working directory, so be careful where you call it from.

```
git clone https://github.com/rairizarry/murders.git
```

[8] https://github.com/rairizarry/murders

ISTUDY



Taylor & Francis

Taylor & Francis Group

http://taylorandfrancis.com

ISTUDY

Index

A

Accessor operator `$`, 20, 23
Adding or transformin variables, 73
Add-on packages, ggplot2 package, 118
`adjust` argument, 121
Advanced Unix, 285
  arguments, 286
  commands, 289–290
  environment variables, 288
  executables, 289
  file manipulation in R, 290
  getting help, 286–287
  permissions and file types, 289
  pipes, 287
  shells, 288
  wild cards, 287–288
`aes` function, 111–112
Aesthetic mappings, 111, 116
Alpha blending, 138
Anchors, 240
Annotations, 116–118
Anti join, 204–205
`anti_join` function, 204
API. *see* Application programming interface
Application programming interface (API), 227
  authentication and authorization, 227
  data formats, 227
  endpoints, 227
  httr2 package, 228–229
  keys, 227
  methods, 227
  rate limits, 227
  requests and responses, 227
  types and concepts, 227–228
`apply` function, 50
Approximately normal, 101
`arrange` function, 61
ASCII, 83
`as.data.table` function, 72
`as.Date` function, 210
`as.IDate` function, 215
`as.integer()` function, 22
`as.ITime` function, 215
`as_tibble` function, 64
Auto-complete functionality of R, 21

B

Barplots, 120
Base R, 10, 83–84
Bell curve, 100
`between` function, 66–67
Bimodality, 169
Binary files, 82–83
Binding, 205
  columns, 205–206
  by rows, 206
`bind_rows` function, 206
Bland-Altman plot, 146
Bounded quantifiers, 240–241
`boxplot` function, 38, 103
Boxplots, 102–103, 121, 171

C

Cascading style sheets (CSS), 223
`case_when` function, 66–67
Categorical data, 95
Categorical variable, 95
`cbind` function, 205
`cd` command, 280–281
`c()` function, 17
Change directory, 280
Character classes, 239–240
Class *character,* 21
`class` function, 19
`clean_names()` function, 198
Coercion, 29
Color blind, 144
Column-wise subsetting, 72–73
Commands, Unix, 279
  `cd`, 280–281
  `cp` (copying files), 284
  examples, 281–282

ISTUDY

`less` (looking at a file), 284
`ls`, 279
`mkdir`, 279–280
`mv` (moving files), 283–284
`rm` (removing files), 284
`rmdir`, 279–280
Conditional expressions, 44–46
Continuous variables, 95
`cp` (copying files) command, 284
CSS. *see* Cascading style sheets
CSS selectors, 225

D

Data analysis, 273
Database APIs, 227
Data distributions, 95
  Boxplots, 102–103
  case study, 96, 104–105
  distributions, 96–97
    histograms, 97–99
    normal distribution, 100–102
    smooth density, 99–100
  stratification, 103
  variable types, 95–96
data exploration, 59
data extraction from web, 221
  application programming interface (API), 227
    httr2 package, 228–229
    types and concepts, 227–228
  HTML, scraping, 221–222
    CSS selectors, 225
    HTML, 222–223
    rvest package, 223–225
  JSON, 226–227
`data.frame` function, 64
Data frames, 19, 23, 191
`data` function, 191
Data import, 79
  filesystem, navigating and managing, 80
    path names, 81
    relative and full paths, 80
    working directory, 81
  file types, 82
    binary files, 82–83
    encoding, 83
    text files, 82
  parsers, 83
    base R, 83–84
    data.table package, 85
    downloading files, 86
    readr package, 84–85
    readxl package, 85
  spreadsheets, organizing data with, 87
Data.table package, 72, 83, 85–86, 196, 208, 214–215
  .() data.table notation, 72, 75
  `dcast`, 197
  `melt`, 196–197
  refining data tables, 72
    adding or transformin variables, 73
    column-wise subsetting, 72–73
    reference *versus* copy, 73–74
    row-wise subsetting, 74–75
  separating variables, 197
  sorting, 76
  summarizing data, 75
    group then summarize, 76
    multiple summaries, 75–76
Data transformations, 165
  base, 167–168
  log transformation, 165–167
  value or scales, 168–169
Data types, R, 19
  accessor `$`, 20–21
  data frames, 19
  examining objects, 20
  factors, 22–23
  lists, 23–24
  matrices, 24–26
  vectors, 21–22
Data visualization, 37, 52, 91, 93, 95, 102
Data visualization in practice, 155
  case study, 155–157, 181, 183–184
    data, 182–183
    logistic transformation, 182
  comparing distributions, 169–170
    accessing computed variables, 178–180
    boxplots, 171
    examples, 174–178
    ridge plots, 172–174
    weighted densities, 180–181
  data transformations, 165
    base, 167–168
    log transformation, 165–167
    value or scales, 168–169
  faceting, 158–159
    `facet_wrap`, 160
    fixed scales, 160–161

ISTUDY

heatmaps, 184–186
multimodal distributions, 169
scatterplots, 157–158
time series plots, 161–165
Data wrangling, 191
`dcast` function, 197
Defining functions, 46–47
Delimiter, 82
Density plots, 120–121
Dependencies, 11
Directories, 80
and subdirectories, 276
Distributions, 96–97
histograms, 97–99
normal distribution, 100–102
smooth density, 99–100
`download.file` function, 86
Downloading files, 86
dplyr, 47–48, 72
`filter`, 48
functions, 66, 111
package, 52–56, 202, 207
`select` function, 54
dslabs library, 19, 155
dslabs package, 10, 53, 80–81, 83, 86, 260

E

eCDF. *see* Empirical cumulative distribution function
EDA. *see* Exploratory data analysis
Editors, 8
Empirical cumulative distribution function (eCDF), 97
Encoding, 83
`excel_sheets` function, 85
Exploratory data analysis (EDA), 93
Extracting variables with pull, 60

F

`facet_grid` function, 158, 160
Faceting, 155, 158–159
`facet_wrap`, 160
fixed scales, 160–161
`factor` function, 22
Fetch and merge, 300
`file.path` function, 81
Filesystem, 275
directories and subdirectories, 276
home directory, 276–278
paths, 278–279
working directory, 278
`filter` function, 47–48, 54
Find command, 290
Flow control, 44
For-loops, 48–49
`fread` function, 86
Frequency table, 97
`fromJSON` function, 229
Full join, 204
`full_join` function, 204
Full path, 80
Functionals, 49–50

G

`gapminder` dataset, 155
Gapminder Foundation, 155–157
Gaussian distribution, 100
`geom_abline` function, 117
geom_boxplot, 121
`geom_density` function, 178
`geom_label` function, 112
`geom_line` function, 162
`geom_point` function, 110–113, 162
geom_raster, 121
geom_text, 112
`geom_text` function, 112
`geomtextpath` package, 164
Geom_tile, 121
`geom_X` function, 110
`get_dups` function, 198
`getwd` function, 81
`ggplot`, 100, 163
ggplot2 package, 52, 93, 107, 155
add-on packages, 118
advantage, 107
aesthetic mappings, 111
annotations, 116–118
behavior, 135
categories as colors, 114–115
geometries, 119
barplots, 120
boxplots, 121
density plots, 120–121
histograms, 120
images, 121–122
geometry, adding, 110–111
global aesthetic mappings, 112–113
graph, components of, 108–109
aesthetic mapping, 108
data, 108
geometry, 108

ISTUDY

grids of plots, 122
initializing an object with data, 109–110
layers, 112
non-aesthetic arguments, 113–114
scales, 116
terminology, 108
updating ggplot objects, 115
`ggplot` function, 109, 112
ggrepel package, 118
ggthemes package, 118
Git, 273, 291
basics, 296
add, 297–299
clone, 300–301
pull, 300
push, 299–300
working directory, 296–297
.gitignore, 301
local repo, 292
remote in, 296
repository, 292
initial setup, 295–296
in RStudio, 301–305
staging area, 292
upstream repo, 292
working directory, 292
works, 292
Git Bash Unix directory, 278
GitHub, 6, 10, 126, 273, 291
accounts, 293
Git, connection with, 294–295
repositories, 293–294
GitHub repo, 303
Global aesthetic mappings, 112–113
Google Sheets, 87
Graphing package, 52
grep command, 290
Grids, 107
of plots, 122
`group_by` function, 59–60, 62
Grouped data frame, 59
Group variability, 133

H

Heatmaps, 184–186
`help` function, 16
`hist` function, 37–38
Histograms, 97–99, 120
Home directory, 276–278
HTML (hyper text markup language), 222
HTML, scraping, 221–222
CSS selectors, 225
HTML, 222–223
rvest package, 223–225
html_node, 224
`html_nodes` function, 225
`html_text` function, 224
httr2 package, 228–229

I

`ifelse` function, 45, 66
If-else statement, 44
`image` function, 38–39
Images, 121–122
`%in%`, 36
Indexing, 34, 52
`%in%`, 36
logical operators, 35
`match`, 36
subsetting with logicals, 34–35
vectors, 34
`which`, 36
Infographics, 91
Inner join, 203–204
`inner_join` function, 203
`installed.packages()`, 11
`install.packages`, 15
Integrated development environment (IDE), 5
intersect, 206–207
`intersect` function, 207
ISO-8859-1, 83

J

Janitor package, 198–199
Jason web tokens (JWT), 227
Joins, 202–203
anti join, 204–205
full join, 204
functions, 201–202
inner join, 203–204
left join, 203
right join, 203
semi join, 204
JSON (JavaScript Object Notation), 226–227
jsonlite package, 226–227, 229

K

Key bindings, 7
knitR package, 306, 309

ISTUDY

L
`lapply` function, 50
Latin-1, 83
Lattice, 107
Left join, 203
`left_join` function, 203
`length` function, 46
`less` command, 284
Levels argument, 22
`library()`, 8–11, 15, 52, 107, 109
`list.files` function, 80
Literate programming documents, 309
ln command, 290
`locale()` function, 217–218
Locales, 216
    `locale()` function, 217–218
    in R, 216–217
    wrangling, 218–220
local modes, 169
Local repository, 292
Logical operators, 35
Logistic transformation (`logit`), 142, 181
Lookarounds, 246–247
`ls()`function, 14–15, 279, 286
Lubridate package, 211–214

M
`make_date` function, 213
`mapply` function, 50
Markdown, 309–310
    global execution options, 311
    header, 310
    knitR, 312
    R code chunks, 311
`match`, 36
Matrices, 24
`matrix` function, 24
`max` and `which.max`, 32
Mean-difference plot, 146
`mean` function, 101
Median, 102
`median_min_max`, 60
`merge` function, 209
`min` and `which.min`, 32
`mkdir` command, 279–280
Modes, 169
Multimodal distributions, 169
Multiple modes, 169
Multiple summaries, 58–59
`mutate` function, 53–55
`mv` command, 283–284
my-first-project.Rproj, 308

N
`names` functions, 27, 192
namespaces, 47–48
Nano command, 290
New script, 8
Nodes, 224
Non-aesthetic arguments, 113–114
Normal distribution, 100–102
Not availables (NA), 30
`nrc` lexicon, 267
nudge_x argument, 113
Numeric variable, 95

O
Object, 14
`object.size` function, 215
Open/start command, 290
`order` function, 31–32, 60
Ordinal data, 95
Outliers, 103

P
Parsers, 79, 83
    base R, 83–84
    downloading files, 86
    readr package, 84–85
    readxl package, 85
Path names, 81
Percentiles, 102
Personal access token (PAT), 294
Pipe, 56–57
    operator, 56
`pivot_longer` function, 192–193
`pivot_longer` is `melt`, 196–197
`pivot_wider` function, 193–194, 196
`pivot_wider` is `dcast`, 197
Placeholder, 64–65
`plot` function, 37
`plotly` package, 93
Plots, R
    `boxplot`, 38
    `hist` function, 37–38
    `image`, 38–39
    `plot` function, 37
Plotting functions, 211
Point-and-click analysis software, 5
Point-and-click approaches, 80, 273, 277

ISTUDY

`print`, 14
Private repositories, 293
Productivity tools, 273
Programming, 44
  conditional expressions, 44–46
  defining functions, 46–47
  for-loops, 48–49
  namespaces, 47–48
  vectorization and functionals, 49–50
Projects, 279
Pseudo-three-dimensional plots, 148–149
Public repositories, 293
purrr function, 65
`purrr` package, 52, 65–66
`pwd`, 278

Q
`quantile` function, 58
Quartiles, 102–103

R
`rank`, 32
`rate`, 74
`rbind` function, 206
`RColorBrewer` package, 147
`rda`, 18
`RData`, 10, 18
`readBin` function, 82
read.csv, 83
read.delim, 83
`readLines` function, 82
readr package, 52, 83–85, 217
readr parsers, 84
read.table, 83
readxl package, 83, 85
Recycling, 34
Reference *versus* copy, 73–74
`reframe` function, 58
Regular expressions (regex) language, 235, 238
Relational operators, 22
Relative paths, 80, 278
Renaming levels, 256–257
`reorder` function, 22
`replicate` function, 50
Repositories, 292–293
Reproducible projects, 306
  data science project, 313
    add, commit, and push files using RStudio, 317
    create directories in Unix, 313
    directories, 316
    editting R scripts, 314–315
    Git directory, 316–317
    main analysis file, 315–316
    README file, 316
    RStudio project, 313–314
    saving processed data, 315
  markdown, 309–310
    global execution options, 311
    header, 310
    knitR, 312
    R code chunks, 311
  RStudio projects, 306–309
`req_perform` function, 228
`request` function, 228
`req_url_path_append`, 228
Reshaping data, 192
  data.table, 196
    `pivot_longer` is `melt`, 196–197
    `pivot_wider` is `dcast`, 197
    separating variables, 197
  janitor package, 198–199
  `pivot_longer`, 192–193
  `pivot_wider`, 193–194
  separating variables, 194–196
Restore .RData into workspace at start, 9
ridge plots, 155, 172–174
right join, 203
`right_join` function, 203
rjson package, 227
`rm` (removing files) command, 284
`rmdir` command, 279–280
Root directory, 80
Row-wise subsetting, 74–75
Rproj, 307
RStudio, 6, 63, 303
  environment and history, 6
  global options, changing, 10
  key bindings, 7–8
  Open file, 82
  panes, 6
  running commands while editing scripts, 8–10
  Tools tab, 10
rvest package, 223–225

S
`sapply` function, 50, 64–65
Save workspace to .RData on exit, 9
`scales` function, 116

ISTUDY

`sd` function, 101
`search()`, 48
search and replace, 244–246
SelectorGadget, 225
Selectors, 225
Semi join, 204
`semi_join` function, 204
`separate_wider_delim` function, 195
Separating variables, 194–196, 247
Sequences, 27–28
`setdiff`, 207–208
`setDT`, 74
`setequal` function, 208
Set operators, 206
  intersect, 206–207
  `setdiff`, 207–208
  `setequal`, 208
  union, 207
`setwd` function, 81
`sheet` argument, 85
Sheets, 85
Shell, 274
Show the data principle, 137, 140
Significant digits, 149–150
Sign in button, 293
Slope charts, 145
Smooth density, 99–100
`sort`, 30, 32
`sort` function, 60
Sorting, 30, 60–61, 76
  `max` and `which.max`, 32
  nested sorting, 61
  `order`, 31
  `rank`, 32
  `sort`, 30
  tibbles, 62–63
  top *n*, 61–62
Special characters, 237–239
Spreadsheet, 79
sqrt function, 15
Square root transformation (`sqrt`), 142, 169
Stage box, 303
Standard deviation (SD), 95, 101
Standard normal, 101
Statistical techniques, 262
stats `filter`, 48
Stats package, 47
Stratification, 103
stringi package, 231
String processing, 231
  case conversion, 248
  case studies, 233–235, 248–256
  escaping, 235–236
  regular expressions, 237
    anchors, 240
    bounded quantifiers, 240–241
    character classes, 239–240
    groups, 243–244
    lookarounds, 246–247
    not, 243
    search and replace, 244–246
    separating variables, 247
    special characters, 237–239
    strings, 237
    unbounded quantifiers: *, ?, +, 242
    white space, 241–242
  renaming levels, 256–257
  stringr package, 231–233
  trimming, 248
stringr, 237
stringr functions, 237
stringr package, 231
strings, 237
`str_replace_all` function, 245
Subsetting, 28–29
  with logicals, 34–35
`sum` function, 46
`summarize` function, 57–60, 64
suppressWarnings, 234
`Sys.getlocale()` function, 216
`system.file` function, 80
`Sys.time()` function, 213

T

Tables, joining, 201
  binding, 205
    columns, 205–206
    by rows, 206
  with data.table, 208
  joins, 202–203
    anti join, 204–205
    full join, 204
    inner join, 203–204
    left join, 203
    right join, 203
    semi join, 204

ISTUDY

set operators, 206
intersect, 206–207
`setdiff`, 207–208
`setequal`, 208
union, 207
`tapply` function, 50
tar command, 290
`tempdir` function, 86
`tempfile` function, 86
Terminal, 275
Text analysis, 260
case studies, 260–262
sentiment analysis, 266–270
text as data, 262–266
Textdata package, 266
Text files, 82
Third variable, encoding, 146–147
`tibble` function, 64
Tibbles *versus* data frames, 63–64
Tidy data, 52–53
tidyr package, 192, 195
`tidytext` function, 266
tidytext package, 262, 266
tidyverse packages, 11, 52, 84, 192, 206, 223
conditionals, 66
`case_when` function, 66–67
`between` function, 66–67
pipe, 56–57
placeholder, 64–65
`purrr` package, 65–66
refining data frames, 53
adding columns, 53–54
column-wise subsetting, 54–55
row-wise subsetting, 54
transforming variables, 55
sorting, 60–61
nested sorting, 61
tibbles, 62–63
top *n*, 61–62
summarizing data, 57
extracting variables with pull, 60
`group_by` function, 59–60
multiple summaries, 58–59
`summarize` function, 57–58
tibbles *versus* data frames, 63–64
tidy data, 52–53
Time series plots, 155, 161–165
Token, 262
`top_n` function, 61–62
Transformations, 141–142, 155
Trimming, 248
`tstrsplit` function, 197

U

Unbounded quantifiers: *, ?, +, 242
Unicode, 83
Union, 207
Unix, 274
advanced Unix, 285
arguments, 286
commands, 289–290
environment variables, 288
executables, 289
file manipulation in R, 290
getting help, 286–287
permissions and file types, 289
pipes, 287
shells, 288
wild cards, 287–288
case study, 285
commands, 279
`cd`, 280–281
`cp` (copying files), 284
examples, 281–282
`less`, 284
`ls`, 279
`mkdir`, 279–280
`mv`, 283–284
`rm`, 284
`rmdir`, 279–280
filesystem, 275
directories and subdirectories, 276
home directory, 276–278
paths, 278–279
working directory, 278
naming convention, 274–275
overview, 274
terminal, 275
Unix shell, 273
Unnest_tokens, 262
Upstream repo, 292
`Users` directory, 280
UTF, 83

V

`vapply` function, 50
Vector arithmetics, 32–33
beware of recycling, 34
rescaling, 33
two vectors, 33–34

ISTUDY

Vectorization and functionals, 49–50
Vectors, 21, 26
    creating vectors, 26
    names, 27
    sequences, 27–28
    subsetting, 28–29
Version control system, 273, 297
visual cues, 143–144
    encoding data using, 126–129

W
Web services, 227
Weighted densities, 180–181
`which`, 36
White space, 241–242
Working directory, 80–81, 292, 296
Workspace, 14–15

Y
`ymd` function, 212

ISTUDY