

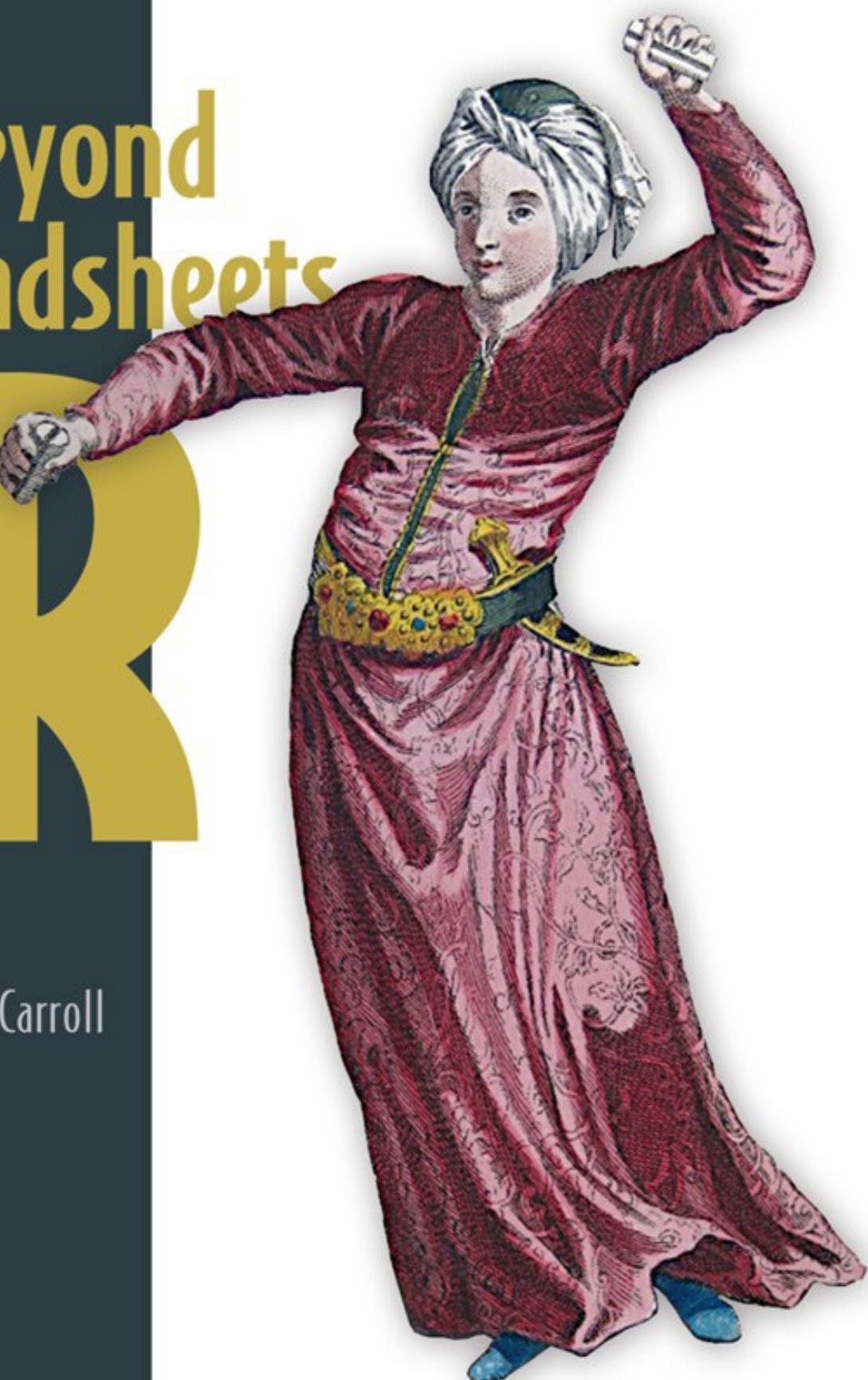
A beginner's guide to R and RStudio

# Beyond Spreadsheets

with

R

Dr. Jonathan Carroll



MANNING

## *about the cover illustration*

---

The figure on the cover of *Beyond Spreadsheets with R* is captioned “Habit of a Turkish Dancer in 1700.” The illustration is taken from Thomas Jefferys’ *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic.

Thomas Jefferys (1719–1771) was called “Geographer to King George III.” He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a map maker sparked an interest in local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this collection. Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late 18th century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries.

The diversity of the drawings in Jefferys’ volumes speaks vividly of the uniqueness and individuality of the world’s nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It’s now often hard to tell the inhabitants of one continent from another. Perhaps, trying to view it optimistically, we’ve traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it’s difficult to tell one computer book from another, Manning celebrates the inventiveness and initiative of

the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys' pictures.

# *contents*

---

*Titlepage*

*Copyright*

*preface*

*acknowledgments*

*about this book*

Who needs this book?

How to read this book

*Formatting*

*Structure*

Getting started

Where to find more help

More about this book

Book forum

*about the author*

*about the cover illustration*

*Chapter 1: Introducing data and the R language*

1.1 Data: What, where, how?

1.1.1 *What is data?*

1.1.2 Seeing the world as data sources

1.1.3 Data munging

1.1.4 What you can do with well-handled data

1.1.5 Data as an asset

1.1.6 Reproducible research and version control

1.2 Introducing R

1.2.1 The origins of R

1.2.2 What R is and what it isn't

1.3 How R works

1.4 Introducing RStudio

1.4.1 Working with R within RStudio

1.4.2 Built-in packages (data and functions)

1.4.3 Built-in documentation

1.4.4 Vignettes

1.5 Try it yourself

Terminology

Summary

## Chapter 2: Getting to know R data types

2.1 Types of data

2.1.1 Numbers

2.1.2 Text (strings)

2.1.3 Categories (factors)

2.1.4 Dates and times

2.1.5 Logicals

2.1.6 Missing values

2.2 Storing values (assigning)

2.2.1 Naming data (variables)

2.2.2 Unchanging data

2.2.3 The assignment operators (<- vs. =)

2.3 Specifying the data type

2.4 Telling R to ignore something

2.5 Try it yourself

Terminology

Summary

## *Chapter 3: Making new data values*

3.1 Basic mathematics

3.2 Operator precedence

3.3 String concatenation (joining)

3.4 Comparisons

3.5 Automatic conversion (coercion)

3.6 Try it yourself

Terminology

Summary

## *Chapter 4: Understanding the tools you'll use: Functions*

4.1 Functions

4.1.1 Under the hood

4.1.2 Function template

4.1.3 Arguments

4.1.4 Multiple arguments

4.1.5 Default arguments

4.1.6 Argument name matching

4.1.7 Partial matching

4.1.8 Scope

4.2 Packages

4.2.1 Installing packages

4.2.2 How does R (not) know about this function?

4.2.3 Namespaces

4.3 Messages, warnings, and errors, oh my!

4.3.1 Creating messages, warnings, and errors

4.3.2 Diagnosing messages, warnings, and errors

4.4 Testing

4.5 Project: Generalizing a function

4.6 Try it yourself

Terminology

Summary

## Chapter 5: Combining data values

5.1 Simple collections

5.1.1 Coercion

5.1.2 Missing values

5.1.3 Attributes

5.1.4 Names

5.2 Sequences

5.2.1 Vector functions

5.2.2 Vector math operations

5.3 Matrices

5.3.1 Naming dimensions

5.4 Lists

5.5 data.frames

5.6 Classes

5.6.1 *The tibble class*

5.6.2 *Structures as function arguments*

5.7 Try it yourself

Terminology

Summary

## Chapter 6: Selecting data values

6.1 Text processing

6.1.1 *Text matching*

6.1.2 *Substrings*

6.1.3 *Text substitutions*

6.1.4 *Regular expressions*

6.2 Selecting components from structures

6.2.1 *Vectors*

6.2.2 *Lists*

6.2.3 *Matrices*

6.3 Replacing values

6.4 data.frames and dplyr

6.4.1 *dplyr verbs*

6.4.2 *Non-standard evaluation*

6.4.3 *Pipes*

6.4.4 *Subsetting data.frame the hard way*

6.5 Replacing NA

6.6 Selecting conditionally

6.7 Summarizing values

6.8 A worked example: Excel vs. R

6.9 Try it yourself

6.9.1 Solutions — no peeking

Terminology

Summary

## *Chapter 7: Doing things with lots of data*

7.1 Tidy data principles

7.1.1 The working directory

7.1.2 Stored data formats

7.1.3 Reading data into R

7.1.4 Scraping data

7.1.5 Inspecting data

7.1.6 Dealing with odd values in data (sentinel values)

7.1.7 Converting to tidy data

7.2 Merging data

7.3 Writing data from R

7.4 Try it yourself

Terminology

Summary

## *Chapter 8: Doing things conditionally: Control structures*

8.1 Looping

8.1.1 Vectorization

8.1.2 Tidy repetition: Looping with purrr

8.1.3 *for loops*

8.2 Wider and narrower loop scope

8.2.1 *while loops*

8.3 Conditional evaluation

8.3.1 *if conditions*

8.3.2 *ifelse conditions*

8.4 Try it yourself

Terminology

Summary

## Chapter 9: Visualizing data: Plotting

9.1 Data preparation

9.1.1 *Tidy data, revisited*

9.1.2 *Importance of data types*

9.2 ggplot2

9.2.1 *General construction*

9.2.2 *Adding points*

9.2.3 *Style aesthetics*

9.2.4 *Adding lines*

9.2.5 *Adding bars*

9.2.6 *Other types of plots*

9.2.7 *Scales*

9.2.8 *Facetting*

9.2.9 *Additional options*

9.3 Plots as objects

9.4 Saving plots

9.5 Try it yourself

Terminology

Summary

## *Chapter 10: Doing more with your data with extensions*

10.1 Writing your own packages

10.1.1 Creating a minimal package

10.1.2 Documentation

10.2 Analyzing your package

10.2.1 Unit testing

10.2.2 Profiling

10.3 What to do next?

10.3.1 Regression

10.3.2 Clustering

10.3.3 Working with maps

10.3.4 Interacting with APIs

10.3.5 Sharing your package

10.4 More resources

Terminology

Summary

## *Appendix A: Installing R*

Windows

Mac

Linux

From source

## *Appendix B: Installing RStudio*

Installing RStudio

Packages used in this book

Appendix C: Graphics in base R

Index

List of Figures

List of Tables

List of Listings

# *Beyond Spreadsheets with R*

---

## ***A beginner's guide to R and RStudio***

DR. JONATHAN CARROLL



MANNING  
Shelter Island

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

©2018 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

Development editor: Jenny Stout  
Project editors: Kevin Sullivan, Janet Vail  
Copy editor: Corbin Collins  
Proofreader: Tiffany Taylor  
Technical proofreader: Hilde Van Gysel  
Typesetter: Happenstance Type-O-Rama

Cover designer: Marija Tudor

ISBN 9781617294594

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – SP – 23 22 21 20 19 18

## *preface*

---

Data is everywhere, and it's used in practically every industry in one way or another. One of the most common ways to interact with data, whether numbers or text, is with spreadsheet software. This approach offers several useful features: presenting data in a tabular view, allowing calculations to be performed using those values, and producing summaries of data. What spreadsheets *don't* tend to provide is a way to do this repeatedly, reproducibly, or programmatically (without clicking or copying and pasting). Spreadsheets can be great for displaying data (including limited data summaries); but when you want to do something truly powerful with data, you need to go beyond them to a programming language.

Data munging—manipulating raw data—is a cornerstone of data science. Munging techniques include cleaning, sorting, parsing, filtering, and pretty much anything else you need to do to make data truly useful. They say 90% of data science is preparing the data, and the other 90% is actually doing something with it. Don't underestimate how important it is to carefully prepare data; analysis interpretations hinge on getting this step right.

Using a programming language to perform data munging means the things you do to your data are recorded, can be reproduced from the raw source, and can be inspected later—even changed, if necessary. Trying to do this from a spreadsheet means either writing down which button to press when, or a broken link between output and input.

I love using R. It's useful in many ways. I never thought a language could be so flexible that it could calculate a t-test one moment and then request an Uber the next. Every word of this book has been processed by R code; the inline results were

generated by actual R code and brought together using a third-party R package (`knitr`). I use R for the vast majority of my work, both data munging and analysis, which over the years has varied from estimating fish abundances to assessing genetic factors in cancer drug trials. I could not have done any of these things if I was limited to working in a spreadsheet program.

Over the course of reading this book, you'll learn enough of the ins and outs of the R programming language to be able to take the data you're interested in and produce an analysis well beyond what you'd be able to accomplish with a spreadsheet.

**NOTE** A message to those of you who have obtained a pirated copy of this book. Copyright infringement is commonly justified by those who partake in it by the notion that "no one loses anything." That's true. But only the infringer gains anything. Many, many hours went into the writing and publication of this book, and without a formal sale involved, any gain you receive from reading this book goes unnoticed and unappreciated. If you have an unofficial copy of this book and have found it useful, please consider buying a legitimate copy, either for yourself or for someone else you think might benefit from it.

## *acknowledgments*

---

I would like to thank Manning Publications for the opportunity to write this book, in particular the large team behind the scenes working to bring it all together, including my editor, Jenny Stout, and the production team of Kevin Sullivan, Janet Vail, and Tiffany Taylor and technical proofreader Hilde Van Gysel. I also thank the dedicated pool of reviewers who provided invaluable feedback during the book’s development, including: Anil Venugopal, Carlos Aya Moreno, Chris Heneghan, Daniel Zingaro, Danil Mironov, Dave King, Fabien Tison, Irina Fedko, Jenice Tom, Jobinesh Purushothaman, John D. Lewis, John MacKintosh, Michael Haller, Mohammed Zuhair Al-Taie, Nii Attoh-Okine, Stuart Woodward, Tony M. Dubitsky, and Tulio Albuquerque.

I’d also like to thank the overwhelmingly helpful communities on Stack Overflow and Twitter (under the #rstats hashtag) and give a special mention to the Asciidoctor team, who have made a fantastic publishing toolchain.

I am eternally grateful to the members of the diverse and supportive R community, the majority of whom voluntarily contribute packages to improve and extend the language. The feedback, suggestions, comments, and discussions I’ve had regarding the contents of this book from reviewers, Twitter followers, and colleagues have helped shape the book into what it is today, and for that I thank each of them.

The maintainers of the R packages mentioned in this book deserve special recognition. The tidyverse of packages has transformed the way I use R and has made working with data much simpler. Producing the code output for this book wouldn’t have been possible without the `knitr` package, and for that I am most thankful.

I would like to thank my wife and children for their support while I wrote this book over the course of around 2 years, without which I would surely have gone mad.

Last but not least, I owe a great deal to the team behind the R language itself. This is open source software, available at no cost to its users. The team's tireless efforts toward continually maintaining and improving this extensive project are greatly appreciated. Their citation can be found from R via the `citation()` function, which produces the following:

R Core Team (2017). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.

# *about this book*

---

## ***Who needs this book?***

You do, of course. Given that you’re reading this, I’m guessing that you have some data (stored as a spreadsheet, perhaps) and aren’t quite sure what to do with it. That’s fine; great, even. Maybe you want to learn something from your data. Maybe you want to find a new way to interact with it. Maybe you want to make a picture out of it. All great goals, but I’m also guessing you want to learn how to do some programming for the first time.

I’m not going to assume you know how to program already, or that you are familiar with the jargon. Perhaps you’ve already picked up a few programming books and been scared off by how fast they fly through the introductory material trying to get you up to speed on every nuance of the way that particular language works. Not here. We’ll take things slow and work on a lot of examples together so that by the time we get to the end you’ll be comfortable with doing what *you* want to do with *your* data.

I’m also not going to even mention statistics. That’s a topic for someone else to cover. If you don’t have a background in statistics, don’t worry; it’s not a requirement here. We’ll be looking at R programming, not statistics (which it, at least, is very good at).

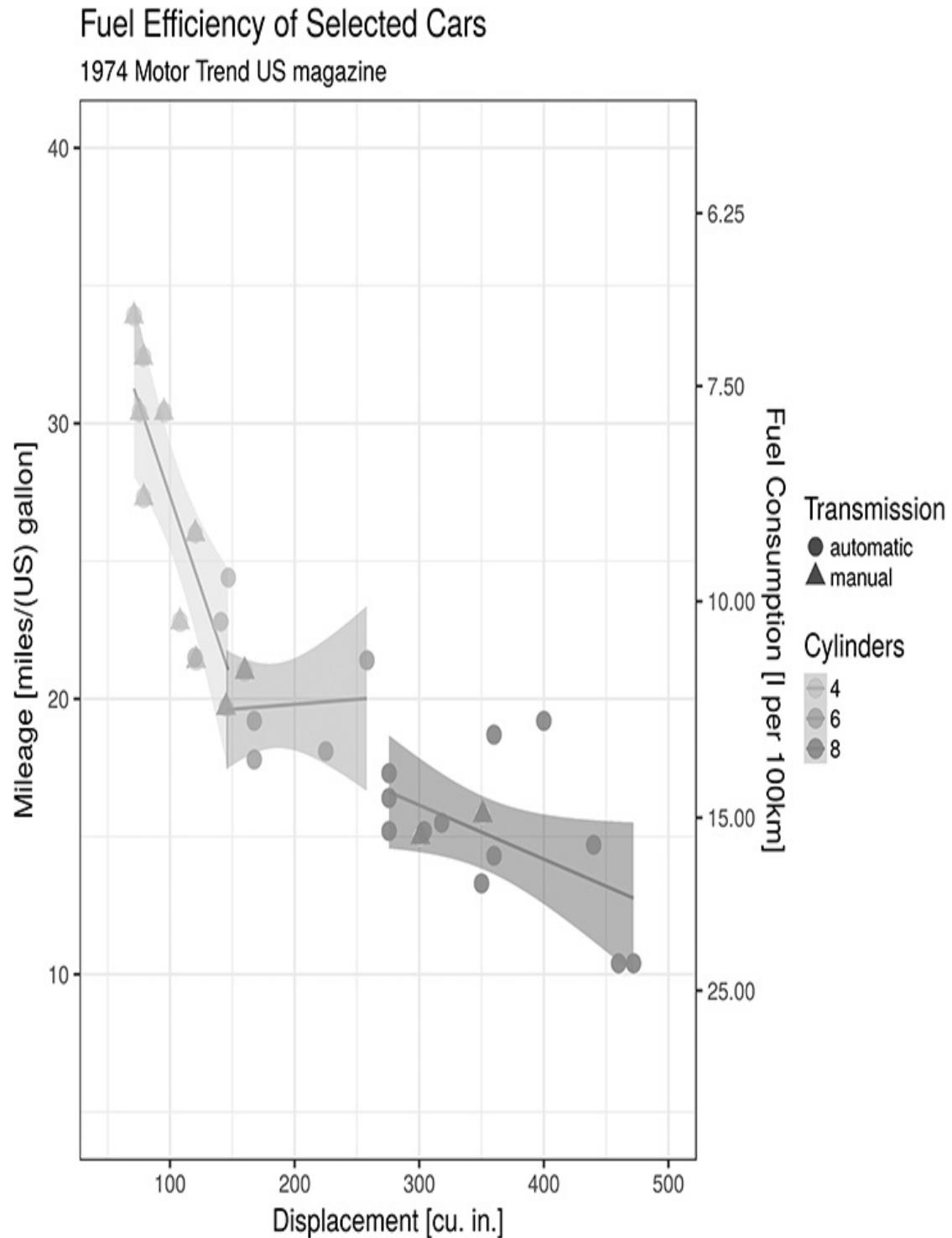
By the time you’ve finished reading this book, you should have a broad understanding of programming and how you do it with the R language; how data can be investigated, interrogated, and used to gain insights; and how to set yourself up for a robust, reproducible workflow that uses data to strengthen your conclusions.

You’ll see how to take a small dataset and transform it into

meaningful, publication-quality graphics with far more flexibility than any spreadsheet software can offer. With just a dozen commands, you can turn the data shown in [figure 1](#) (the `mtcars` dataset already available from within R, as shown in the RStudio data viewer) into the graphic in [figure 2](#).

	▲ mpg ▾	▲ cyl ▾	▲ disp ▾	▲ hp ▾	▲ drat ▾	▲ wt ▾	▲ qsec ▾	▲ vs ▾	▲ am ▾	▲ gear ▾	▲ carb ▾
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

**Figure 1** The mtcars dataset, available from within R, as viewed in the RStudio data viewer. This data was extracted from the 1974 *Motor Trend US* magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).



**Figure 2.** This visualization of the `mtcars` dataset plots the mileage (`mpg`, as well as fuel consumption in transformed units) against the engine displacement (`disp`) of the 32 vehicles, grouped both by the number of cylinders (`cy1`) and distinguished by their transmission (`am`), along with a linear fit to each cylinder group's data. This is achieved, formatting

and all, in just a dozen lines of R code.

## ***How to read this book***

I present each chapter to you in a no-nonsense manner; I cover what's important and what's likely to become an issue if you're not careful. I can't cover *every* way to approach a problem, and I may not do it necessarily the same way that other texts approach problems. But I try to show you what I consider to be the best approach first and back that up with some alternatives that you may be likely to also encounter in other reading. The goal here is to make you a competent and productive R user, which may mean showing you how to do things the slow way (as well as the fast way).

## **FORMATTING**

New terms and definitions are shown in *italics* when they are first mentioned. Code samples and data values are printed in a monospace font, either inline (for mentions of code) such as `str(mtcars)` or in code blocks for examples you should try yourself, such as this one:

```
myData <- head(mtcars, n = 2)
```

When a code sample produces output, this is shown below the input with the prefix #> and you should generally expect to see the same if you run the code yourself. The output for the vast majority of examples has been generated by R itself in the course of writing this book. Don't worry if you try to run the lines starting with #>; they will be ignored by R:

```
myData
#>          mpg cyl disp  hp drat    wt  qsec vs am
gear carb
#> Mazda RX4     21   6 160 110  3.9 2.620 16.46  0  1
4      4
#> Mazda RX4 Wag 21   6 160 110  3.9 2.875 17.02  0  1
4      4
```

Options that are available via a menu appear as a sequence of

selections to make, such as File > Save > OK. And I tell you plainly which buttons to click and which keys you need to press.

Examples are sometimes shown as blocks of annotated code, like this, which reads some data from a .csv file and calculates the average height value:

```
peopleData <- read.csv(file = "people.csv")  
summary(peopleData)  
mean(peopleData$height)
```

①  
②  
③

①

Reads the data from the .csv file into a data.frame

②

summary() acting on a data.frame returns a column-wise 5-number summary.

③

You can take the mean() of a column of values.

Certain kinds of information are highlighted along the way:

**NOTE** When a piece of information is particularly critical or important, it will be presented in a block like this one. Such blocks also indicate additional information, historical curiosities, or other notes.

**CAUTION** R won't always stop you from doing something you didn't intend. In fact, sometimes it will seem to be actively trying to catch fire. Where fires are easily started, they're pointed out like this to help you avoid them.

**TIP** There are typically many ways to solve a problem using R, and I only discuss the simplest in any detail here. Where a better solution exists (but requires more information), I note it like this and try to give you enough information to go find out more yourself.

In some cases, code blocks are not accompanied by output, because the code does not actually run. These code blocks are for illustration purposes only. Where output *is* shown, you should expect to get similar results when you run the code.

Errors produced by R begin with the word `Error`. You'll see lots of these in the code in this book. The precise wording of the error may differ slightly between versions. Please take care when entering blocks of code containing one of these errors, as that output cannot be parsed by R.

Throughout the book I'll also show you what a spreadsheet equivalent starting point might look like. I will use LibreOffice, which looks like [figure 3](#), but the concepts will usually extend to Excel, Google Sheets, or whichever spreadsheet software you usually use.

	A	B
1	1	x
2	2	y
3	3	z
4		

**Figure 3** An example of cells selected in LibreOffice (Linux)

## STRUCTURE

As we progress through the book together, there will be lots of examples that I hope you will work through. Don't just read them—run them on your computer yourself and see if you get the same answers. Then try a variation on the example and see if you get the result you expect. If you get something different, that's great! It means you've found something to learn from, and your

next task will be to understand why the result is what it is.

I will try to progressively build up your knowledge of the relevant programming and R-specific terms, so don't be afraid to go back and revise if something seems unfamiliar.

## **Getting started**

Here's what you will need:

- This book
- A computer
- A desire to learn something

Really, that's about it. R is a *free* (as in speech—openly available—and as in beer—it costs nothing) language, and we'll be using more free software to interact with it. You will probably need an internet connection to download the (free) software, but after that the majority of examples will work offline.

Follow along with the examples as they appear. Try different values and see if you get the result you expect. Break things and try to understand what happened. It's very difficult to end up in a situation that can't be resolved by restarting R, so feel free to experiment.

This book won't necessarily direct you toward how to solve *your* specific problems, but it should give you enough of a comprehension of the language and its ecosystem for you to begin working out what other tools you might need to use. If you're working in genomics, there's a good chance you'll need some more advanced tools provided by the Bioconductor suite of packages: [www.bioconductor.org](http://www.bioconductor.org). Many of the concepts and structures used there extend from those you'll learn about in this book (though I don't cover those here).

## **Where to find more help**

Stack Overflow (<https://stackoverflow.com>) is an immensely

useful source of information under the `r` tag, but it's frequently overrun with poorly researched questions and thankless responses. Take the time to figure out if your question has already been answered (which happens regularly, given how many questions have been asked) before insisting that someone else solve your problem.

If all else fails, typing what terms you do know and `r` or `rstats` into a search engine (such as Google) tends to produce some useful results more often than not.

The R Weekly site (<https://rweekly.org>) provides a weekly summary of the most interesting R posts from around the web. R-bloggers (<https://r-bloggers.com>) provides a syndication of many popular R-related blogs and has fresh content daily. Follow along with some of these that align with your interests, and you're bound to come across some useful tips.

Finally, reach out to your local community, either in person (try <https://meetup.com>) or online (Twitter, `#rstats`).

## ***More about this book***

This book was written in the AsciiDoc plain-text markup language using emacs and RStudio. The R code herein was evaluated using a custom package library defined via the `switchr` R package and intertwined among the source using the `knitr` R package.

The session information describing the environment defining this custom library is as follows:

```
#> setting  value
#> version   R version 3.4.3 (2017-11-30)
#> system    x86_64, linux-gnu
#> ui        X11
#> language  en_AU:en
#> collate   en_AU.UTF-8
#> tz        Australia/Adelaide
#> date      2018-01-23
#>
#> package      * version  date      source
```

#> assertthat	0.2.0	2017-04-11	CRAN	(R 3.4.3)
#> backports	1.1.2	2017-12-13	CRAN	(R 3.4.3)
#> base	* 3.4.3	2017-12-01	local	
#> bindr	0.1	2016-11-13	CRAN	(R 3.4.3)
#> bindrcpp	0.2	2017-06-17	CRAN	(R 3.4.3)
#> broom	0.4.3	2017-11-20	CRAN	(R 3.4.3)
#> cellranger	1.1.0	2016-07-27	CRAN	(R 3.4.3)
#> cli	1.0.0	2017-11-05	CRAN	(R 3.4.3)
#> colorspace	1.3-2	2016-12-14	CRAN	(R 3.4.3)
#> commonmark	1.4	2017-09-01	CRAN	(R 3.4.3)
#> compiler	3.4.3	2017-12-01	local	
#> crayon	1.3.4	2017-09-16	CRAN	(R 3.4.3)
#> crosstalk	1.0.0	2016-12-21	CRAN	(R 3.4.3)
#> curl	3.1	2017-12-12	CRAN	(R 3.4.3)
#> data.table	1.10.4-3	2017-10-27	CRAN	(R 3.4.3)
#> datasauRus	* 0.1.2	2017-05-08	CRAN	(R 3.4.3)
#> datasets	* 3.4.3	2017-12-01	local	
#> devtools	* 1.13.4	2017-11-09	CRAN	(R 3.4.3)
#> digest	0.6.14	2018-01-14	CRAN	(R 3.4.3)
#> dplyr	* 0.7.4	2017-09-28	CRAN	(R 3.4.3)
#> evaluate	0.10.1	2017-06-24	CRAN	(R 3.4.3)
#>forcats	* 0.2.0	2017-01-23	CRAN	(R 3.4.3)
#> foreign	0.8-67	2016-09-13	CRAN	(R 3.3.1)
#> ggplot2	* 2.2.1	2016-12-30	CRAN	(R 3.4.3)
#> glue	1.2.0	2017-10-29	CRAN	(R 3.4.3)
#> graphics	* 3.4.3	2017-12-01	local	
#> grDevices	* 3.4.3	2017-12-01	local	
#> grid	3.4.3	2017-12-01	local	
#> gtable	0.2.0	2016-02-26	CRAN	(R 3.4.3)
#> haven	1.1.1	2018-01-18	CRAN	(R 3.4.3)
#> here	* 0.1	2017-05-28	CRAN	(R 3.4.3)
#> hms	0.4.0	2017-11-23	CRAN	(R 3.4.3)
#> htmltools	0.3.6	2017-04-28	CRAN	(R 3.4.3)
#> htmlwidgets	* 1.0	2018-01-20	CRAN	(R 3.4.3)
#> httpuv	1.3.5	2017-07-04	CRAN	(R 3.4.3)
#> httr	* 1.3.1	2017-08-20	CRAN	(R 3.4.3)
#> jsonlite	1.5	2017-06-01	CRAN	(R 3.4.3)
#> knitr	* 1.18	2017-12-27	CRAN	(R 3.4.3)
#> lattice	0.20-35	2017-03-25	CRAN	(R 3.3.3)
#> lazyeval	0.2.1	2017-10-29	CRAN	(R 3.4.3)
#> leaflet	* 1.1.0	2017-02-21	CRAN	(R 3.4.3)
#> lubridate	1.7.1	2017-11-03	CRAN	(R 3.4.3)
#> magrittr	1.5	2014-11-22	CRAN	(R 3.4.3)
#> mapproj	* 1.2-5	2017-06-08	CRAN	(R 3.4.3)
#> maps	* 3.2.0	2017-06-08	CRAN	(R 3.4.3)
#> memoise	1.1.0	2017-04-21	CRAN	(R 3.4.3)
#> methods	* 3.4.3	2017-12-01	local	
#> mime	0.5	2016-07-07	CRAN	(R 3.4.3)

#> misc3d	0.8-4	2013-01-25	CRAN	(R 3.4.3)
#> mnormt	1.5-5	2016-10-15	CRAN	(R 3.4.3)
#> modelr	0.1.1	2017-07-24	CRAN	(R 3.4.3)
#> munsell	0.4.3	2016-02-13	CRAN	(R 3.4.3)
#> nlme	3.1-131	2017-02-06	CRAN	(R 3.4.0)
#> openxlsx	4.0.17	2017-03-23	CRAN	(R 3.4.3)
#> parallel	3.4.3	2017-12-01	local	
#> pillar	1.1.0	2018-01-14	CRAN	(R 3.4.3)
#> pkgconfig	2.0.1	2017-03-21	CRAN	(R 3.4.3)
#> plot3D	*	1.1.1	2017-08-28	CRAN (R 3.4.3)
#> plyr	1.8.4	2016-06-08	CRAN	(R 3.4.3)
#> psych	1.7.8	2017-09-09	CRAN	(R 3.4.3)
#> purrr	*	0.2.4	2017-10-18	CRAN (R 3.4.3)
#> R6	2.2.2	2017-06-17	CRAN	(R 3.4.3)
#> Rcpp	0.12.15	2018-01-20	CRAN	(R 3.4.3)
#> readr	*	1.1.1	2017-05-16	CRAN (R 3.4.3)
#> readxl	1.0.0	2017-04-18	CRAN	(R 3.4.3)
#> reshape2	*	1.4.3	2017-12-11	CRAN (R 3.4.3)
#> rex	*	1.1.2	2017-10-19	CRAN (R 3.4.3)
#> rio	*	0.5.5	2017-06-18	CRAN (R 3.4.3)
#> rlang	*	0.1.6	2017-12-21	CRAN (R 3.4.3)
#> rmarkdown	*	1.8	2017-11-17	CRAN (R 3.4.3)
#> roxygen2	*	6.0.1	2017-02-06	CRAN (R 3.4.3)
#> rprojroot	1.3-2	2018-01-03	CRAN	(R 3.4.3)
#> rstudioapi	0.7	2017-09-07	CRAN	(R 3.4.3)
#> rvest	0.3.2	2016-06-17	CRAN	(R 3.4.3)
#> scales	0.5.0	2017-08-24	CRAN	(R 3.4.3)
#> shiny	1.0.5	2017-08-23	CRAN	(R 3.4.3)
#> stats	*	3.4.3	2017-12-01	local
#> stringi	1.1.6	2017-11-17	CRAN	(R 3.4.3)
#> stringr	*	1.2.0	2017-02-18	CRAN (R 3.4.3)
#> switchr	*	0.12.6	2017-11-07	CRAN (R 3.4.1)
#> testthat	*	2.0.0	2017-12-13	CRAN (R 3.4.3)
#> tibble	*	1.4.1	2017-12-25	CRAN (R 3.4.3)
#> tidyverse	*	0.7.2	2017-10-16	CRAN (R 3.4.3)
#> tidyverse	*	1.2.1	2017-11-14	CRAN (R 3.4.3)
#> tools	3.4.3	2017-12-01	local	
#> utils	*	3.4.3	2017-12-01	local
#> withr	2.1.1	2017-12-19	CRAN	(R 3.4.3)
#> xml2	1.1.1	2017-01-24	CRAN	(R 3.4.3)
#> xtable	1.8-2	2016-02-05	CRAN	(R 3.4.3)

Details for installing the specific versions of these packages are provided in appendix C. The code for the examples in the book is located at <https://github.com/BeyondSpreadsheetsWithR/Book>. There is also an issue tracker where people can link directly to

the R code in which they find an issue:

<https://github.com/BeyondSpreadsheetsWithR/Book/issues>.

The source code is also available from the publisher's website at  
[www.manning.com/books/beyond-spreadsheets-with-r](http://www.manning.com/books/beyond-spreadsheets-with-r).

## ***Book forum***

Purchase of *Beyond Spreadsheets with R* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/beyond-spreadsheets-with-r>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

## *about the author*

---



**Ewa Jermakowicz**

JONATHAN CARROLL holds a PhD in theoretical astrophysics from the University of Adelaide, Australia, and is currently working as an independent contractor providing R programming services in data science. He contributes packages to R, is a frequent contributor of answers on StackOverflow, and is an avid science communicator.

# 1

## *Introducing data and the R language*

---

### **This chapter covers**

- Why data analysis is important
- How to make your analysis robust
- How and why R works with data
- RStudio: Your interface to R

You have your data, and you want to start doing something awesome with it, right? Brilliant! I promise you, we'll get to that as soon as we can. But first, let's take a step back. Telling you to dive right in now would be like handing you a pile of different timbers, pointing you toward the workshop, and telling you to make some furniture. It's a good idea to first understand both the materials and the tools you're about to use.

We'll go through what *data* means in general — to you and to those who may potentially inherit your data — because if you don't fully comprehend what you already have, then building on that won't be useful (and at worst will be flat out wrong). Poorly preparing data merely delays dealing with it properly and grows your *technical debt* (making things easier now, but later making it necessary to pay back that time when you have difficulties working with poorly formed data).

We'll discuss how to set yourself up for a rigorous analysis (one that can be repeated) and then begin working with one of the best data analysis tools available: the R programming language. For now, let's go through what it means to "have some data."

## **1.1 Data: What, where, how?**

I said you have some data that you want to do something with, which wasn't a very precise statement. That was intentional. I guarantee you have some data even if you don't realize it. You may be thinking that *data* is exclusively whatever is stored in your Excel file, but data is much more than that. We all have data, because it's everywhere. Before you go analyzing your own data, it's important to recognize its structure (both as you understand it, and as R will) so that you begin with a solid foundation of what it means to *have some data*.

### **1.1.1 WHAT IS DATA?**

Data exists in many forms, not just as numbers and letters in a spreadsheet. It may also be stored in a different file type, such as comma-separated values (CSV), as words in a book, or as values in a table on a web page.

**NOTE** It's common to store comma-separated values in a .csv file. This format is particularly useful because it's plain text — values separated by commas. We'll return to why that's useful in section 1.1.6.

Data may not be stored at all — *streaming* data comes as a flow of information, such as the signal your TV picks up and processes, your Twitter feed, or the output from a measuring device. We can store this data if we want to, but often we want to understand the flow as it's happening.

Data isn't always pretty (in fact, most times it's dirty,

mundane, and seemingly uninteresting), and it isn't always in the format we want. Having some tools on hand to manage data is a powerful advantage and is critical to achieving a reliable goal, but that's only useful if you know what your data represents before you do anything further with it. "Garbage in, garbage out" warns that you can't perform an analysis on terrible data and expect to get a meaningful result. You may very well have tried to evaluate a calculation in Excel only to have the result show up as #VALUE! because you tried to divide a number by some text, even though that "text" looked like numbers. The types of your values (text, numbers, images, and so on) are themselves pieces of data with possible meanings behind them, and you'll learn how to best make use of them.

So what is "good data"? What do the values you have represent?

### ***1.1.2 SEEING THE WORLD AS DATA SOURCES***

We experience the world through our senses — touching, seeing, hearing, tasting, smelling, and generally absorbing life around us. Each of those input channels handles available data, and our brains process them, mixing the signals together to form our picture of the world in a brilliantly complex way that we constantly take for granted.

Every time you use any of your senses, you're taking a measurement of the world. How bright is the sun today? Is a car approaching? Is something burning? Is there enough coffee left in the pot for another cup? We construct measuring tools to make life easier for us and handle some of the data consistently — thermometers to measure temperatures, scales to measure weights, rulers to measure lengths.

We go a step further and create more tools to summarize that data — car instrument panels to simplify the internal measurements of the engine; weather stations to summarize temperature, wind, and pressure. With the digital age, we now

have an overload of data sources at our disposal. The internet provides data on virtually any and all aspects of the world we might be interested in, and we create more tools to manage these — weather, finance, social media, the number of astronauts currently in space ([www.howmanypeopleareinspace.com](http://www.howmanypeopleareinspace.com)), lists of episodes of *The Simpsons*, all available at our disposal. The world is truly made up of data.

That's not to say the data is in any way finite. We constantly add to the available sources of data, and by asking new questions we can identify new data we want to obtain. Data itself also generates more data. *Metadata* is the additional data that describes some other data — the number of subjects in a trial, the units of a measurement, the time at which a sample was taken, the website from which the data was collected. All these are data too and need to be stored, maintained, and updated as they change.

You interact with data in various ways all the time. One of the greatest achievements of the World Wide Web has been to gather, collate, and summarize our data for us in more easily digestible forms. Think about how you would have requested a taxi 20 years ago, before the rise of smartphones and the app ecosystem. You'd look up the phone number of a taxi company, phone them, tell the dispatcher where you were or would be, where you wanted to go, and what time you wanted to be picked up. The dispatcher would send out the request to all drivers, one of whom would accept the request. At the end of your journey, you'd pay with cash or a card transaction and receive a receipt.

Now, with the digital connections between devices, continuous internet access, and GPS tracking, that process simplifies to opening a ride-share app, entering your destination, and receiving a fare estimate, because your phone already knows where you are. The ride-share program receives this data and selects an appropriately close/available driver, exchanges your contact details in case anyone needs them, and routes the driver to you. At the end of your journey, your account is charged the

appropriate amount, and a receipt is emailed to you.

In both cases, the same data flowed between all the parties. In the latter, fewer people needed to be involved because the computer systems have access to the relevant data. Your phone communicates with the ride-share server, your phone communicates with the GPS system to locate itself, and the ride-share server communicates with a payment server to authorize payment and the email server to send the receipt.

At every point along the way, various data can be collected (anonymously, where required) and saved for later analysis. How many people requested rides to the airport this month? What was the average distance travelled? What was the average wait time? Do people request more expensive trips from Apple or Android devices? Some of this was available previously, but it has never been easier to aggregate and compare.

Many businesses open up access to third-party developers using an *application programming interface* (API) so that the data can be more systematically accessed. For example, Uber has an API that allows software to ask for fare estimates or ride histories (with authentication, to approved accounts). This is how your phone app is able to communicate with the Uber servers. Sure enough, someone has written an R package to work with this API, meaning you can include data direct from Uber in your analysis, or (in theory) request a ride direct from R.

**NOTE** Good software has a documented way to interact with it so that users and the software are able to communicate clearly and effectively. This can describe requests that can be sent to a server (and the expected responses) or just how a function should be used (and the expected return value).

### 1.1.3 DATA MUNGING

*Data munging* refers to the cleaning up and preparation of data.

Most data collected isn't ready to be used in an analysis or presentation. Usually there are inputs to validate, summaries to calculate, values to combine or remove, or restructuring to perform. This is a commonly overlooked aspect of using data for science, but it's of vital importance. Failing to properly handle data can lead to difficulty working with it and, worse, incorrect conclusions drawn from it.

The terms *data munging*, *data wrangling*, *data science*, *data analysis*, *data hazmat*, and many others are all names for more or less the same thing, with different emphases and different trajectories depending on where the data is coming from or going to. Most analyses (be they elaborate, sophisticated regressions, or simple visualizations) begin with some form of data munging. Often that's merely reading the data into software, in which case some of the handling is performed on your behalf with *assumptions* (these values are treated as dates, these as words, and so on). Having the power to control how that handling is performed can be essential when those assumptions are broken, or when you want to treat your data in a particular way.

Any time you have groups of records in your data, whether years, patients, animals, colors, vehicles, or anything else, and you need to treat them differently (color a line a certain way, only include records in an average of similar things, calculate how a quantity has changed between groups), you'll perform data munging because you need to allocate records to a particular group somehow. Any other transformation, cleaning, or processing of the data also counts toward data munging. It quickly becomes apparent that a large portion of any analysis can (or should) involve a lot of data munging if its conclusions are to be trusted.

### **1.1.4 WHAT YOU CAN DO WITH WELL-HANDED DATA**

I hope it's clear by this point that data is potentially of great

importance. It is routinely more than just numbers in a table. Medical data often represents real human lives and the effect a particular intervention has had, be that lifesavingly positive or tragically negative. These effects aren't always immediately obvious to someone viewing them from a given perspective, so it's the role of the data analyst (professional or incidental) to extract patterns from data in order to make a decision.

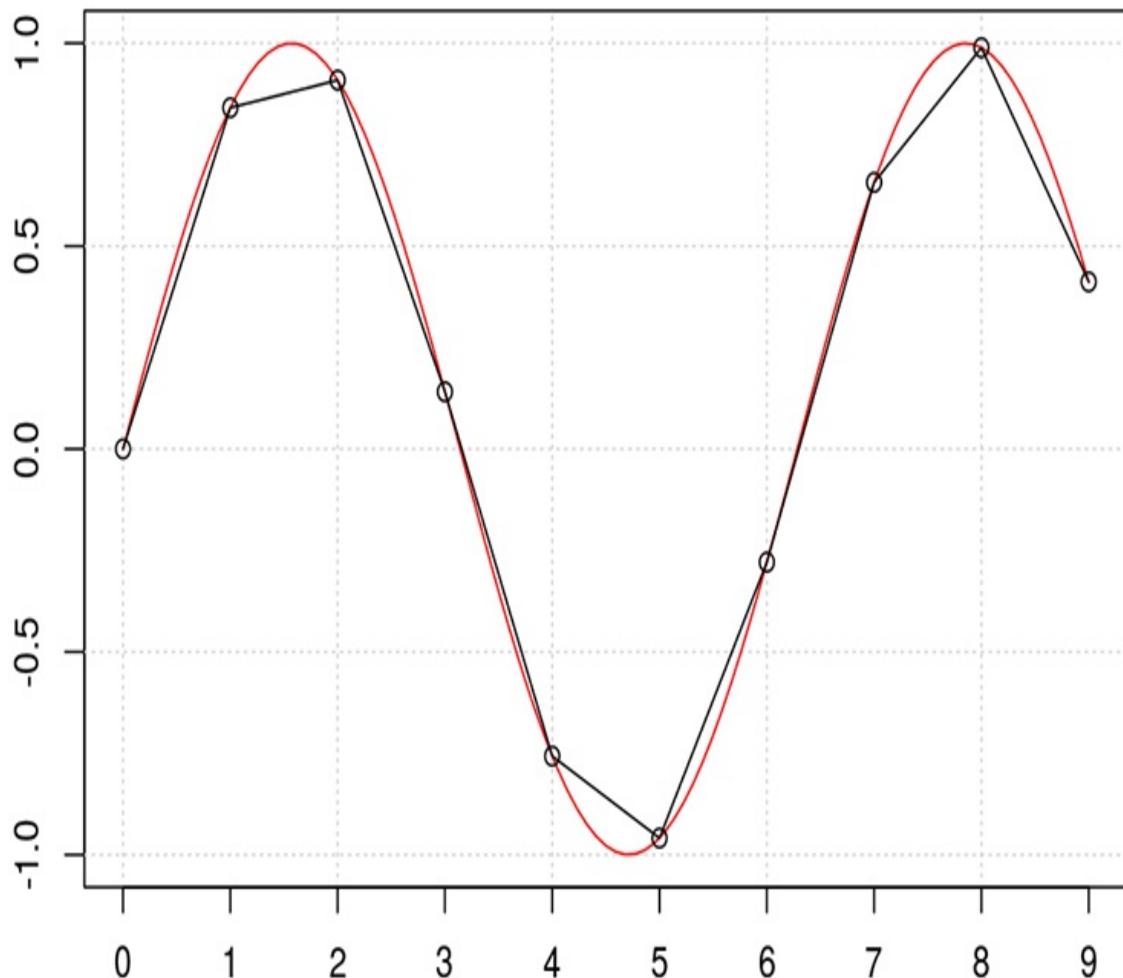
Analysis of data is often useful in extracting nonobvious patterns. For example, although you may recognize a pattern to the sequence

```
#> 2 4 6 8 10 12 14 16 18 20
```

(counting by twos), it may not be so clear what the pattern is in the following data

```
#> 0.000 0.841 0.909 0.141 -0.757 -0.959 -0.279 0.657  
0.989 0.412
```

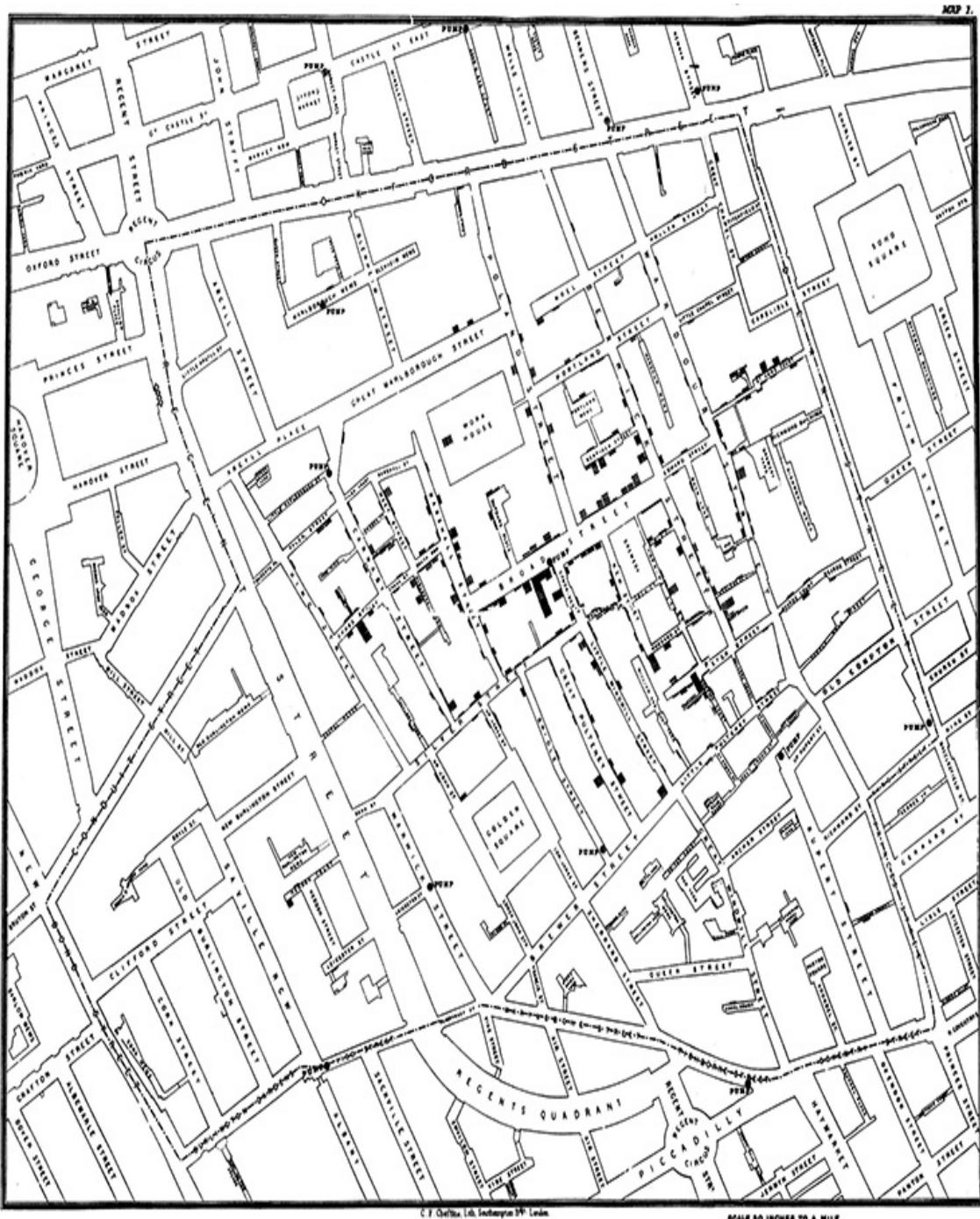
until you visualize the data (which was generated with a `sin()` function), as shown in [figure 1.1](#). Having the right tools at hand to analyze our data means we can identify hidden patterns, forecast new information, and learn from the data.



**Figure 1.1** A pattern emerges. These points were generated with a `sin()` function at the values 0, 1, ..., 9. The smooth `sin()` function is also plotted here.

A classic example of data analysis is that of John Snow and the 1854 Broad Street cholera outbreak in London. People were dying by the hundreds within a particular district at a time when sewerage infrastructure was all but nonexistent and the understanding of infectious diseases was highly limited. By carefully examining the locations of the cholera cases, John Snow was able to infer that the common link between them appeared to be that their closest source of water was a particular pump on Broad Street. Once the pump was disabled, cases of cholera diminished significantly. In this case, the data was in plain sight — the locations of cholera cases — but the pattern and

connection weren't immediately apparent. See figure 1.2.

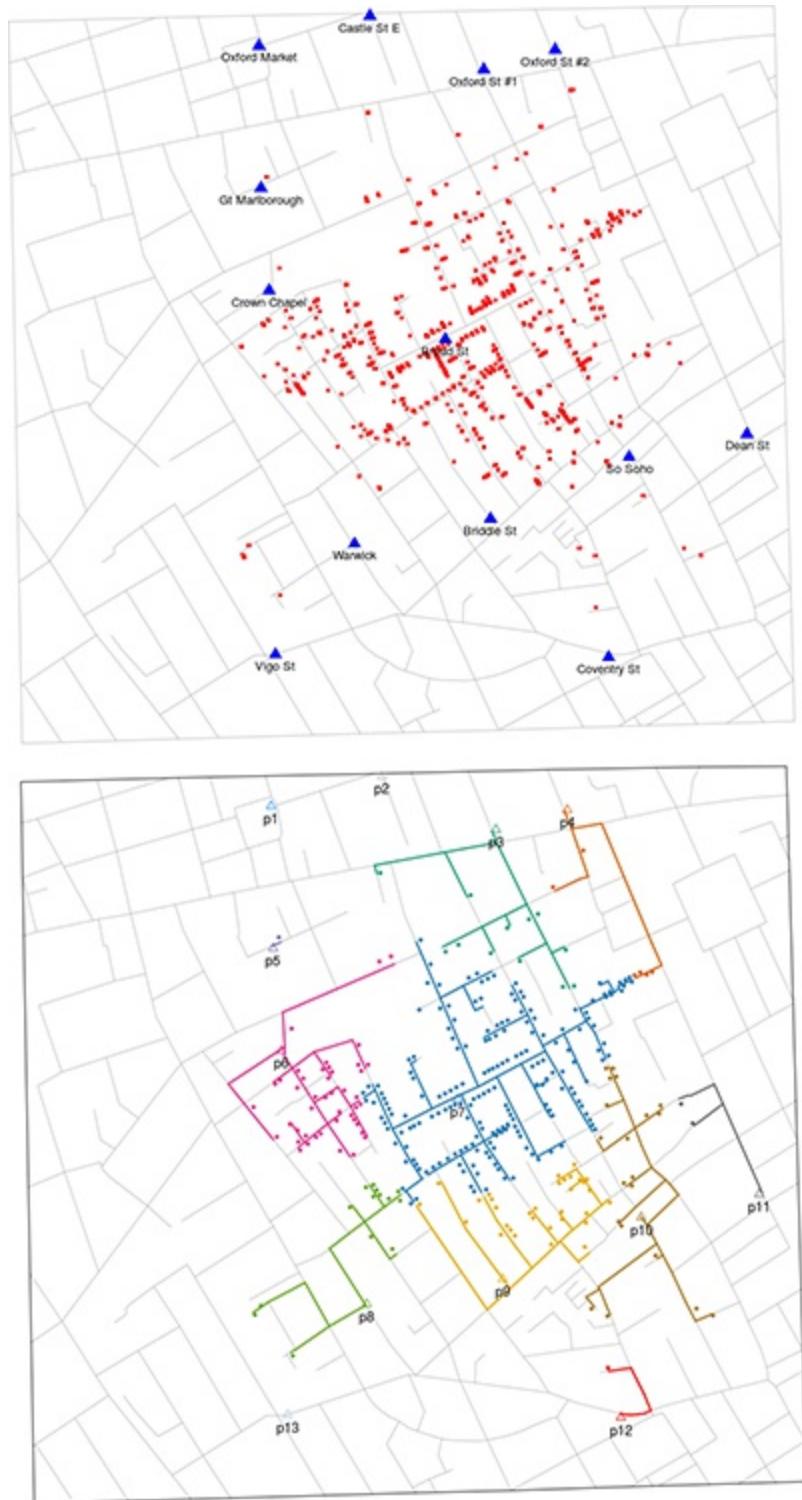


**Figure 1.2** The Broad Street Cholera Map, by John Snow (public domain), via Wikimedia Commons. Dots indicate pump locations, and cases of cholera are marked with stacked bars along streets.

Perhaps unsurprisingly, several R packages are available to

interact with this data. The raw data can be found in the `HistData` package, and a further graphical analysis in the `cholera` package, resulting in [figure 1.3](#).

Sometimes a spreadsheet program such as Excel or Libre Office is a sufficient tool for this purpose. Viewing some tabular numbers together, sorting them, and perhaps plotting them as a bar chart are all easily achieved in a wide range of software applications. When we want to interact with the data in a more structured, formal, reproducible, and rigorous manner, though, we turn to a programming language. R is an excellent choice.



**Figure 1.3** Further analyses of the Broad Street cholera data produced using the `HistData` (top) and `cholera` (bottom) R packages

### 1.1.5 DATA AS AN ASSET

Data is powerful, because data is information we learn from. We are rarely in a situation where we have no access to any data whatsoever (not just digital), but different data comes with different responsibilities.

Weather data is relied on by many to plan their day, be they fishermen figuring out how far they should venture from the relative safety of the shore, or a winegrower growing wine-making grapes planning the likelihood of overnight frost ruining their crop. Weather forecasts aren't the rawest source of data but compiled summaries produced by digesting rawer sources of measurement.

Similarly, financial analysts provide assessments of the stock markets and insights into the likely day-to-day movements of critical investments. These too are generated from models that ingest high-frequency measurements of the current state of the market and provide higher-level summaries that are easier to grasp and act on.

In each of these cases, there are custodians of data who are relied upon: those who make available the raw measurements in a predictable and robust manner. Should these sources of data become corrupted, either the raw or processed sources, then those further down the chain are unable to provide reliable processing of that data, and there are potential consequences to follow. I would personally put a lot less faith in a weather report if I knew that the raw readings had been entered by hand into a spreadsheet and the forecast created by someone remembering in which order the buttons needed to be pressed, which cells needed to be copied over to another sheet, and which rows needed to be selected to be included in the calculation.

The motivation behind highlighting this fact will hopefully stay with you throughout this book — we are all part of a data chain, and if we don't take care with the data while it's in our possession, then all steps that follow are subject to failure in ways that won't necessarily be apparent to those who query our data. We therefore seek to produce robust, reproducible, and

transparent processing of any and all data we access and release back into the wild.

Although a thorough description of reproducible research requires significantly more resources to fully detail, the following guideline will serve you well for now:

- Document how, when, and from where you obtained your data.
- Provide commentary on any decisions you make during your handling of the data.
- Leave raw sources of data unchanged — anything you create along the way should be documented and reproducible, ideally without your involvement.

**TIP** Reproducible research is key to trusting your results, even if they don't seem to be of great significance. It may very well be only yourself looking at the results in a year, but knowing how you produced new data is just as important as what the data tells you.

Being able to trace back through the changes that a dataset has undergone is invaluable to justifying an analysis. You may end up with a plot of median income per capita for European countries, but can you tell how the scaling was performed from that? Was the data filtered for overseas income? Was the data a sample or a census? Without knowing what steps went into the analysis, the final result raises unanswerable questions.

It's critical that any analysis you perform starts with the *right* data, data that's collected in an appropriate manner and that addresses the question you're asking. That question needs to be the *right* one too; otherwise you won't learn what you're hoping to.

Far better an approximate answer to the right question, which is often vague, than the exact answer to the wrong question, which can always be made precise.

—John Tukey, founding chairman of the Princeton Statistics Department

With the right data and the right question in hand, how do you go about keeping track of everything? For that, you need to be able to properly handle not just the data and the code, but *how* it changes over time.

## 1.1.6 REPRODUCIBLE RESEARCH AND VERSION CONTROL

Have you ever received a file with a filename like mydata\_final\_Thurs20May\_phil\_fixed\_final\_v2.xlsx? Not the most succinct name, but it hints at something much worse — that multiple copies of the file are floating around, each with a different version of the data, most of which is out of date due to some corrections or updates, and with unknown changes between versions. If someone presented a graph produced from one of these files, could you be certain which version it came from? Or if presented with the most recent graph and the one that preceded it, could you tell what had changed?

The answer is to not rely on the filename to store the versioning information (which it is poorly suited to do). Instead, *version control systems* (VCS) can keep track of the changes so that you (and any collaborators you are working with)

- Are always up to date with the latest version of all files
- Can review the changes between versions
- Can roll back to any previous version

Part of this is aided greatly by using *plain-text* files (such as .txt, .R, and .csv) because a version control system can literally compare the lines of two versions and show you what's changed. Using *binary* files (such as .docx and .pdf) makes this more difficult to extract, but doesn't make it useless:

- *Plain-text file* — A file that stores its contents as numbers, letters, and punctuation, and as such can be opened in a text

editor. Information in a plain-text file can be read into any system, and because it has no formatting, there's no ambiguity about what each symbol represents or how to read it. This doesn't preclude storage of formatting, but that too needs to be in plain text, such as a markup language that uses tags around values like `<b>bold text</b>`, or a markdown language that uses inline modifiers like `**bold**`.

- *Binary file* — A file that stores its contents in binary (zeroes and ones) to be interpreted by suitable software. It's not readable in a text editor but has the advantage that it can encode the formatting of data, including a variety of different formats including sounds, images, or video.

I won't cover specific VCS options here, but you should find one that works for you. Some popular options include the following:

- Git (using GitHub/GitLab/Bitbucket)
- Subversion (also known as SVN)
- Mercurial

Each of those has a learning curve of its own but pays for itself the first time you need to undo a swath of changes or deletions.

Another great benefit of version control is that you can openly share (if you like) the code that describes what you've done with your data so that someone interested (possibly another data analyst, possibly yourself six months from now) can reproduce your work because they have the inputs and the analysis steps.

Have you ever completed working on some data and become worried that perhaps you haven't saved your file, and that you might have to go through all those steps again (if you can even remember what they were)? If you can't remember what steps you performed after just completing them, how can you trust that you did them correctly? How could someone else? By working with a *script* of commands, which you can think of as a log of exactly what you told the computer to do, you're keeping a record of the analysis steps, and someone should be able to reach

the same conclusions as you did if they start with the same data.

It's not uncommon for data to require updates, and when that happens it's easy to spot the difference between people who follow reproducible research methods and those who don't. After weeks of data processing and number crunching, someone will notice that there was a typo in column 12 of the third data set and send out an updated file: `data3_fixedTypo.csv`.

The benefits of reproducible research are many. The person who *doesn't* follow reproducible research does the following:

1. Deletes all outputs (or saves them elsewhere)
2. Opens up the new data file
3. Performs all of the analysis steps as best as they can remember them
4. Forgets that column 4 needs special treatment
5. Doesn't understand that the final results are more different than they should be

The person who follows reproducible research does the following:

1. Changes the input data filename in their script
2. Reruns the analysis script that contains all the required steps and their documentation
3. Knows that the only thing that has changed this time is the input data update

Many R packages exist for helping us work within reproducible research frameworks, and we'll talk about some of the more common ones later.

With our data at the ready, our questions screaming for answers, and our intentions focused on reproducible research through version control, the only thing we still require is a way to bring it all together to produce some results: the R programming language.

## 1.2 Introducing R

R is a statistical programming language, in that it was made for the purpose of performing statistics calculations, but it has grown to be much more through community contributions. As a general-purpose language, R is flexible enough to work with almost any data you can interact with: stored or streaming, images, text, or numbers.

Like most programming languages, it has a specific *syntax* (way of writing things) that may seem confusing or odd at first, but trust me, you'll get used to it soon enough. Believe it or not, R is one of the more readable languages.

R is used both professionally and recreationally by a fast-growing number of users.<sup>1</sup> Anywhere you find data, there's a good chance you'll find someone working with R. A good metric for the popularity of R is the list of professional users of RStudio (the software we'll use to interact with R), the logos of some of which are shown in figure 1.4.

---

<sup>1</sup> As of 2017, it was ranked sixth in the IEEE Spectrum's top 10 programming languages (<http://mng.bz/z5sN>) and eighth in the TIOBE index of popular programming languages for 2017 ([www.tiobe.com/tiobe-index/](http://www.tiobe.com/tiobe-index/)).

---



**Figure 1.4 Professional users of R (rstudio.com)**

Many other companies use R as part of their data-processing capabilities. Some well-known professional users and their

specific uses include the following:<sup>2</sup>

---

<sup>2</sup> See Deepanshu Bhalla, “List of Companies Using R,” *Data Science Central*, <http://mng.bz/qJ66>.

- *Genentech* — Uses R for data munging and visualization and has ties to the core R developers
- *Facebook* — Uses R for exploratory data analysis and experimental analysis
- *Twitter* — Uses R for data visualization and semantic clustering
- *City of Chicago* — Uses R to build a food-poisoning monitor
- *New York Times* — Uses R for interactive features (such as the Dialect Quiz and Election Forecast) and data visualization
- *Microsoft* — Uses R for XBox matchmaking
- *John Deere* — Uses R for statistical analysis (forecasting crop yields and long-term demand for farming equipment)
- *ANZ Bank* — Uses R for credit-risk analysis

R is widely used in academic research of genetics, fisheries, psychology, statistics, and linguistics, among many others. Amateurs have found plenty of *fun* things to do, such as solving Sudoku puzzles (<https://dirk.shinyapps.io/sudoku-solver>) and mazes (<https://github.com/Vessy/Rmaze>), playing chess (<http://jkunst.com/rchess>), and connecting to online services such as Uber (<https://github.com/DataWookie/ubeR>).

In this section you’ll learn how R does what it does and how you’ll interact with it. As with any new tool, beginning with a proper understanding of the available features can save a lot of time down the road. To fully appreciate some of the quirks of R, we need to go back to the start.

### **1.2.1 THE ORIGINS OF R**

The predecessor of R was the programming language S (for

statistics), developed by John Chambers and colleagues at Bell Labs. This was commercialized in 1993 through an exclusive license as S-PLUS, which was used in a wide variety of disciplines. The community saw significant growth when R was conceived as an open source implementation of the S language, meaning everyday users could both see the underlying structure and build on it. Nonetheless, the new language was backward compatible with S, and much of R's weirdness that remains can be attributed to that still being the case.

In February 2000, the first stable release of R was released by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The foundations of R have since been developed by a group of volunteers: the R-core developers and through proposals submitted by the general public. Development also continues in the form of externally produced add-on packages that are officially hosted on the Comprehensive R Archive Network (CRAN, <https://cran.r-project.org>), of which there were roughly 12,000 by the end of 2017; many more are hosted informally on code-sharing sites such as GitHub.

## **1.2.2 WHAT R IS AND WHAT IT ISN'T**

Classifications among programming languages are plentiful and largely obscure. They're also constantly argued over because their definitions are complex and require a degree in computer science to fully appreciate. Although R (well, S) was originally built for statistics, it can be considered a *general purpose language* (GPL) in that it isn't tied to completing just one single task.

Some languages exist purely to achieve a task within some *domain* (a specific area of interest such as finance, technical drawing, or machine control) and these are referred to as *domain-specific languages* (DSLs). R is much more flexible than that because you can write your code so that it achieves whichever goal you need.

R is not a DSL. It's a language for writing DSLs, which is

something that's altogether more powerful.

One person may have a finance data goal in mind, another may be interested in natural language processing, and someone else may be aiming to predict what decisions a customer will make next. The common link between all these is data, but R is so flexible that it provides a capable mechanism to work within each of these domains.

—Joe Cheng, CTO of RStudio

I'm not going to sell R to you; I think it's a great language that makes many tasks simpler and that has a nice way of doing things, but I won't try to tell you it's the only way to solve your specific problem. It may not even be the *best* way. But by learning a new language, we don't try to shoehorn a solution into a problem; instead we learn more about how languages work, which helps us better identify *how* a problem might be solved, even if that means another language is more suitable. Comparing programming languages is like asking which is better, apples or oranges — as usual, it depends, or maybe it doesn't. A slice of each, please.

## WHAT IT IS

At its most basic level, R is a useful tool for interacting with data. It stores *values* (data) and *functions* (code that interacts with data) as *variables* (names for things) and complex *objects* (structures). In technical terms, R is an *open source, interpreted, general purpose, functional* language:

- *Open source* — The underlying source code can be freely obtained and (if desired) modified.
- *Interpreted* — R doesn't require compiling your code into a standalone program. Some languages require the code to be built into an executable in order to run it.
- *General purpose* — It isn't restricted to doing just one thing in a particular domain.

- *Functional* — It uses functions operating on unchanging data, rather than depending on the current state of the system and modifying data in place.

R can be thought of as a toolbelt. You can add more tools to it if you know where to hang them, you can rearrange them to make them more user-friendly, and you can work with just a few tools or many, depending on your needs. The tools in this sense are *packages*, logical groups of documented functions (code to perform operations on data) that can be called on to produce some output — a graph, more data, a signal to process, a request to a website, or just about anything.

Without packages (and I include the base packages and those installed by default), R is merely a framework with limited capabilities. The true power comes when additional packages build on this framework to create powerful statistical functions and publication-quality graphics that themselves can be extended and modified as required.

## WHAT IT ISN'T

Having a good toolbelt doesn't automatically mean you know how to swing a hammer, or that you'll know the difference between a Phillips and a Torx screw, and having R installed won't mean that all of a sudden your data analysis procedures will become clear.

R will let you do almost anything to your data, be that a wise choice or a completely unjustified one. In some cases, it will warn you that you're doing something you possibly don't want to. At other times, it will silently produce garbage and move on to the next step as if nothing were wrong. That's not entirely R's fault — many believe a good programming language should "do what you say, not what you mean" and should let users decide what's right and what's wrong.

Because of the way R works (we'll get to that shortly), it's not always the fastest method of processing data, though it's

certainly not slow. Depending on your use case, speed may not be an issue at all. Sometimes the overhead of using R is an extra few minutes over some other language, with the trade-off being that R code may be much more usable. Many R packages utilize R's ability to interact with other languages and strike a balance between what's processed with R and what's processed with another, more efficient language (such as C).

## 1.3 How R works

Some programming languages *compile* (build) code into an executable program. That has its advantages and disadvantages, but it's not the way R works. Instead, R is an *interpreted* language in that the computer works with instructions one at a time (a series of these is a *script*), and the results from each instruction are presented (*returned*) to the user.

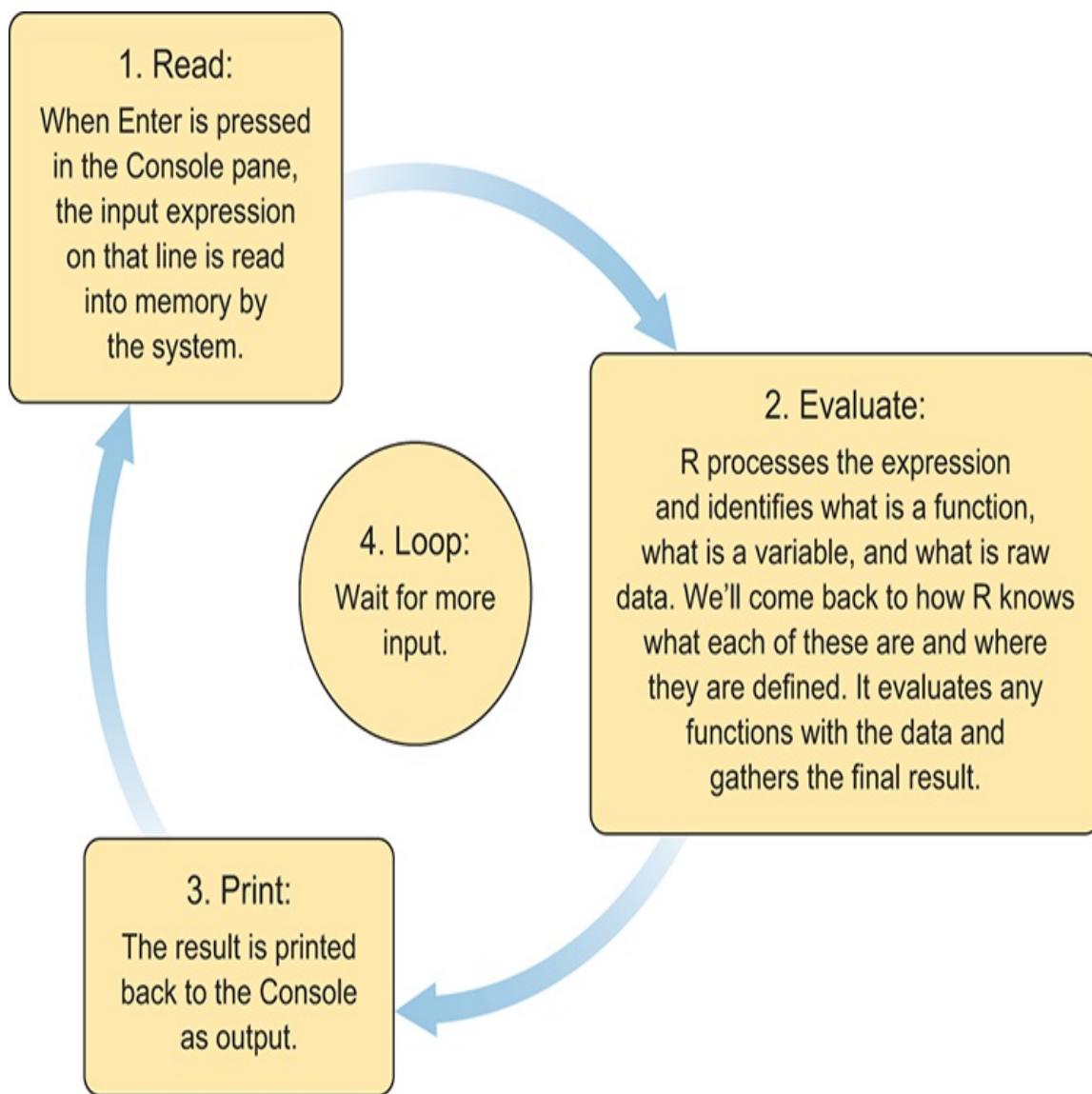
In order to operate in this way, R implements a read, evaluate, print, loop mechanism (REPL), which does exactly what it sounds like. A diagram of this flow is shown in figure 1.5. R waits patiently for your input, and once it is entered it's *read* into the system and *evaluated* (calculations are performed), the result is *printed* back to the Console (if there is any), and the entire process *loops* back to wait for more input.

That may seem like a lot of capability for a language that I just said waits for input before doing anything, and that's because the R *program* (R.exe or the R executable you run to start R) is written mainly in C, which is a *compiled* language (and a very memory efficient one at that). Pressing Enter triggers the C code to perform the REPL operations.

Being an open source language, the source for the code that runs *under the hood* is available for anyone to inspect. The official source for all versions back to R 0.60.1 is available from <https://svn.r-project.org/R/branches>, which means you can see how the various components of R have changed over the years if you like. That's impossible with a proprietary (closed source) program, where the internal workings are only available to those

working on it. With open source software, you can even download the entire source, make changes to it, and compile your own personal version.<sup>3</sup>

<sup>3</sup> The code is licensed under GPLv2, which means you can do whatever you want to it as long as you maintain the attributions of everyone who has worked on it and don't sell it for profit or restrict access.



**Figure 1.5 Read, evaluate, print, loop**

There's also a more accessible read-only mirror hosted by Winston Chang at <https://github.com/wch/r-source>. It's kept in sync with the official source hourly.

If you haven't already done so, install R on your computer. Refer to the instructions in appendix A.

At first you'll issue commands to R one at a time, but eventually you'll want to be able to tell R to do many things in sequence. This is called a *script*, and the allusion to the lines an actor will speak is apt. An R script (typically a file ending with .R or .r) is merely a series of commands, usually one per line but that can be split over many lines, to be read in sequence by the R system and processed. This is particularly different than how a spreadsheet file behaves, where the data in its current state is preserved, but not how it got there. Some of the first lines of the script instruct R how to prepare for the upcoming analysis, followed by how to obtain/read the raw data, then how to process it, and finally how and where to save the results. With this workflow, the analysis can be made reproducible, because armed with the raw data and the processing steps, the results can be reproduced.

R alone is sufficient to process an analysis script, which can be passed to the R processor using the command line. On Windows, depending on your exact version and installation path,<sup>4</sup> from a command line active in the same directory as your script file, you may be able to use the following command:

---

<sup>4</sup> Unless you set the \$PATH environment variable to search this directory.

---

```
C:\Program Files\R\R-3.4.3\bin\R CMD BATCH yourScriptFile.R
```

On a Linux or Mac system,<sup>5</sup> you enter the following at the command line:

---

<sup>5</sup> Assuming the installation directory is in the \$PATH.

---

```
R -f yourScriptFile.R
```

R will start and process the contents of the script file.

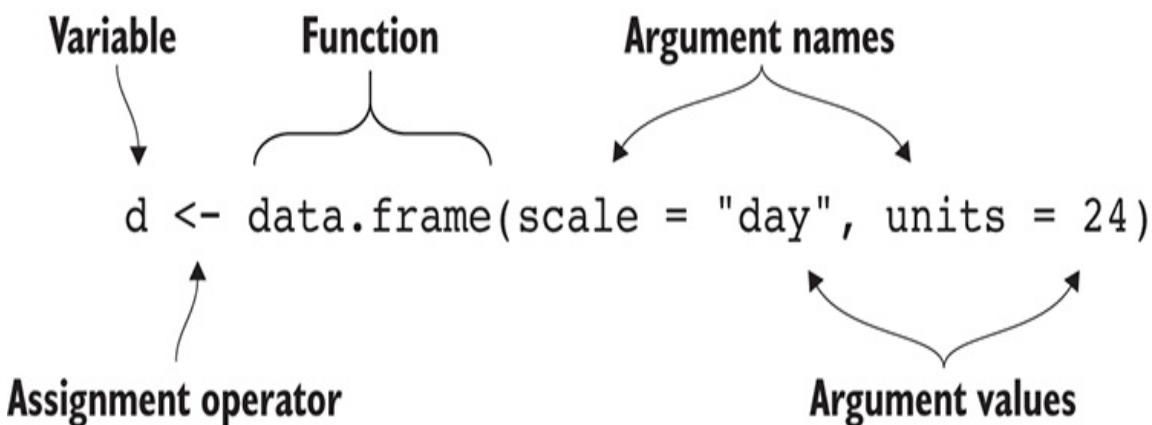
If you use R *interactively* (where you start R yourself and it produces a prompt, awaiting input), you can achieve this same

behavior using the `source()` function:

```
source(file = "~/yourScriptFile.R")
```

The *tilde (~)* in the directory name is a common placeholder for the user's home directory. Your scripts can be placed in any folder; you just need to tell R where to look.

Although you may be familiar with button menus in Excel, R is a command-based language. That means you'll be telling R what to do with *expressions*, pieces of code that perform operations on the data and store the results. Let's take a moment to see what this looks like and some of the names you'll encounter; see [figure 1.6](#). Different types of things may be colored (the syntax highlighting); this helps distinguish different parts of your code.



**Figure 1.6** R code with some terms identified: variable, assignment operator (`<-`), function, and arguments

Some terms used in [figure 1.6](#) may be new to you:

- **Variable** – A name to refer to a piece of data
- **Assignment operator** – Function that stores a value in a *variable*
- **Function** – Some code that interacts with data, *called* (invoked) with an opening ( and a closing ) (parentheses), possibly with *arguments*
- **Arguments** – Options passed to *functions*, separated by commas, possibly as pairs of argument names and argument

values linked by an equals sign (=) — for example, `save = TRUE`

Working with R in this way (reading commands from a saved file) is certainly possible, but to really get a helping hand along the way, we turn to an additional piece of software that wraps around the R system and provides additional functionality: RStudio.

## 1.4 Introducing RStudio

Data is stored on your computer (or some device or drive that your computer can connect to), but interacting with it requires some software to read the data, interpret what you want done to it, and write it to some sort of output or storage (either as values, an image, a sound, or something entirely different).

This can take a wide range of forms:

- Viewing the raw, locally stored data in a text editor such as Notepad or emacs
- Displaying formatted data in a spreadsheet or database program such as Excel, Access, or Google Sheets
- Viewing either unencoded or translated JSON data as it passes over the internet with a browser such as Google Chrome or Internet Explorer
- Using programming software to retrieve and manipulate data

All of those have different abilities in terms of displaying and interacting with data. When it comes to using R for interacting with data, a highly sophisticated and powerful *interactive development environment* (IDE) brings all these abilities together in the form of RStudio. With RStudio, you will be able to view your data in many forms, interact with it, manipulate it, and then store it or distribute it. This IDE features an R-aware text editor for reading/writing scripts, a Console for entering R commands, and best of all, a way to inspect the current state of the Workspace and all the defined variables.

If you haven't already done so, install RStudio on your computer. Refer to the instructions in appendix A.

### **1.4.1 WORKING WITH R WITHIN RSTUDIO**

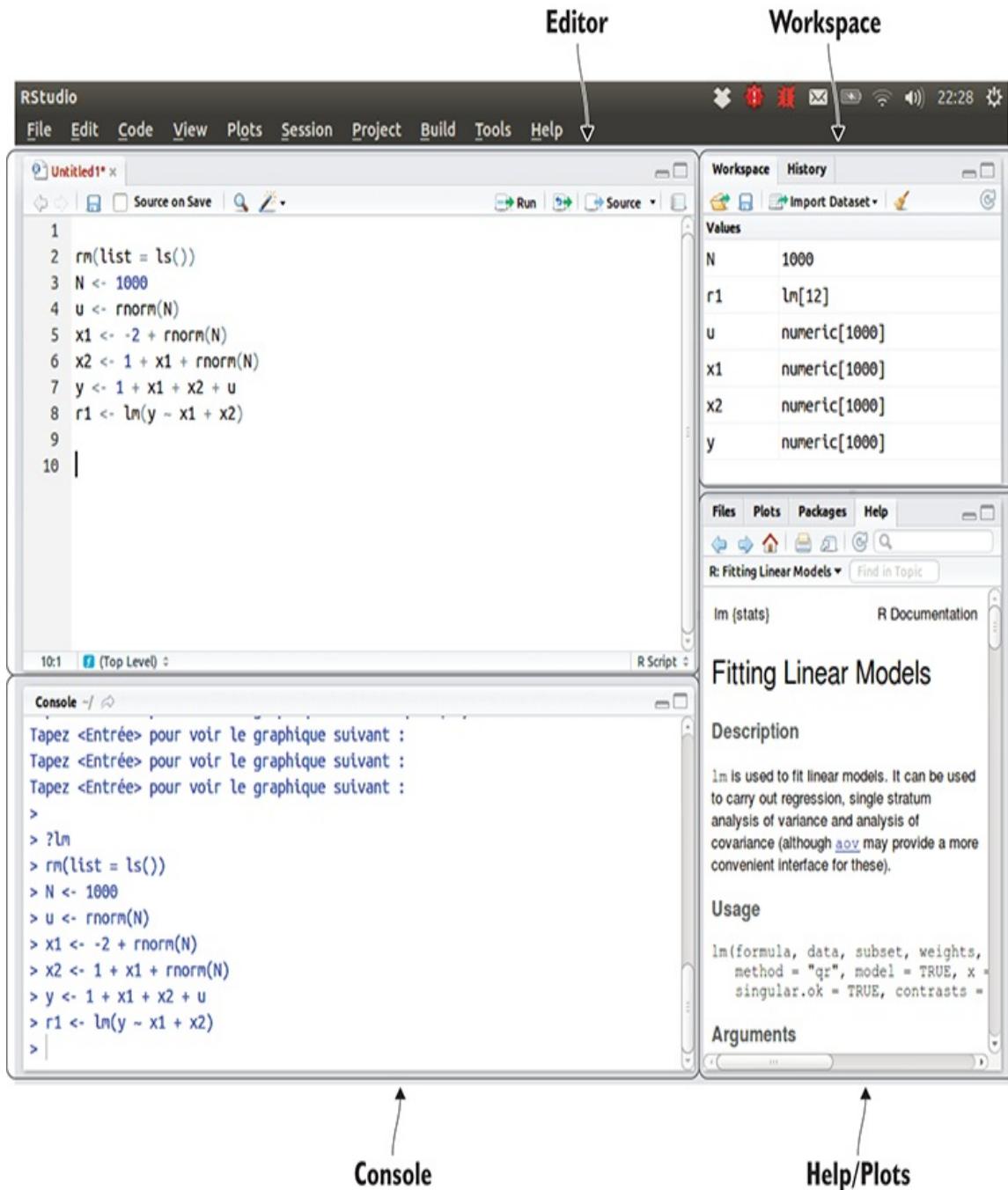
RStudio divides the window into separate *panes* or sections (see figure 1.7<sup>6</sup>). The borders of these can be dragged to expand or contract individual panes, and can be arranged as you prefer by clicking Tools > Global Options > Pane Layout in the menu. Some can also be detached from the main window to be made full-screen.

---

<sup>6</sup> Adapted from a screenshot originally by PAC2 ([www.gnu.org/licenses/agpl.html](http://www.gnu.org/licenses/agpl.html)), via Wikimedia Commons.

The four panes as they appear by default are as follows:

- *Editor* — This is where your scripts are written. A script is a series of commands to be executed in order. When you first open RStudio, the Editor will be empty. Click File > New File > R Script to start a new file/script.
- *Console* — The R prompt as it would appear in a terminal. This is where you enter commands line by line, followed by pressing Enter. Results returned from R are presented here.
- *Workspace* — The values R knows about: your data, or the *variables* you have defined appear in the Environment tab, and the history of which commands you've executed appear in the History tab.
- *Help and Plots* — Depending on which tab you have selected, Help or Plots will display either the documentation for a function or dataset, or the most recent plot produced. It also contains the Packages and Files tabs for listing installed packages and files from your computer, respectively.



**Figure 1.7 RStudio panes as they appear in Ubuntu/Linux**

There are easy ways to switch between these panes. Ctrl-1 moves the cursor to the Editor pane for writing *scripts*. Ctrl-2 moves the cursor to the Console pane for interactive commands. While the cursor is on a function, pressing F1 will bring up the Help menu for that function. There are many other keyboard shortcuts available. Try Alt/Option-Shift-K to bring up an extensive

cheatsheet.

## Alternatives to RStudio

Of course, RStudio isn't the *only* way to use R, though I certainly find it to be the most convenient. If working with a command-line interface is more your style (and you can forego the added benefits RStudio offers), then R works fine within a terminal. R can also be hooked into emacs using the Emacs Speaks Statistics (ESS) emacs package. When you first install R under Windows, you'll also find that RGui is installed, which is a simple graphical interface.

Several alternative graphical interfaces to R are also widely used, such as R Commander and Deducer. For consistency (and because I genuinely believe it to be superior), the remainder of this book assumes you're working within RStudio.

RStudio lends a helpful hand while working with R, but you can certainly do everything you need to in a terminal alone. Your textual interaction with R in that case would still match what will appear in the Console pane of RStudio, which we'll focus on now.

RStudio works nicely with Git and SVN right out of the box. I recommend you read up on that from RStudio directly at <http://mng.bz/1s4F>.

Each time you start an R session (running the R program and working with the R language) either within RStudio or standalone, the Workspace begins empty.<sup>7</sup> If you have the option enabled from the settings (on by default), then the files you had open last time you used RStudio will still appear in the Editor pane. The Workspace pane won't have any objects listed, and the Console will greet you with the following welcome message:

---

<sup>7</sup> Unless you deliberately set the option to start where you left off by loading the Workspace image.

---

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"      ①
Copyright (C) 2017 The R Foundation for Statistical
Computing
Platform: x86_64-pc-linux-gnu (64-bit)      ②
```

R is free software and comes with ABSOLUTELY NO  
WARRANTY. ③

You are welcome to redistribute it under certain  
conditions.

Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.

Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in  
publications.

Type 'demo()' for some demos, 'help()' for on-line help,  
or ④  
'help.start()' for an HTML browser interface to  
help. ④  
Type 'q()' to quit R. ④

> ⑤

①

Version is listed here (and date it was released). Try to keep your  
version as up-to-date as possible, as bug fixes and improvements  
are regularly added. Note, this may require you to update your  
package library also.

②

This has to do with whether you're running Windows, Linux, or  
Mac.

③

You get what you pay for. Although great care is taken to ensure  
things work as expected, there's no formal guarantee of that as

such.

(4)

**Often overlooked, but these are great tips.**

(5)

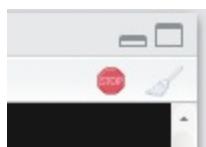
**The prompt symbol > indicates R is ready and waiting for input.**

**NOTE** Since late 2011, R versions have been designated not only by a version number but also by a quaint name, beginning with Great Pumpkin. The names are entirely the whims of a core developer, and though there's no strict structure, they're typically seasonally themed and all are linked to the comic *Peanuts* by Charles M. Schulz. For example, R 3.4.3 (the version used in this book) is nicknamed Kite-Eating Tree.

When the prompt (>) is visible, R is awaiting your next command. Commands can be entered directly into the Console (better for short commands used once) and executed by pressing Enter. Commands can also be built up in the Editor as a script (better for longer analyses and saving your steps); commands can be executed one at a time by pressing Ctrl-Enter (also Cmd-Enter on a Mac) while the cursor is on the relevant line, or many at a time with a highlighted region.

**TIP** If you need to enter more than one command on a single line, you can separate them using a semicolon (;), as in a <- 2; b <- 3. If you begin writing a long command and are part way through — say, you have an opening ( without a closing ) or only half of a calculation such as 2 + — and press Enter, R will assume you mean to insert a line break in the command and will replace the prompt with a + to indicate that it's waiting for the rest of the command, which you can enter as normal. R will join

all the input lines together before executing them. In the Editor pane, commands can be split in the same way (over several lines) and executed either as a whole or in part. Don't worry if you get stuck in this mode. Pressing Enter again will only insert more line breaks. To exit this mode (or cancel entering a command at any time), press Esc to clear the current input and return you to the prompt (>).



**Figure 1.8** Look for this symbol above the Console while R is running.

If you're performing a complex calculation, R will indicate that it's busy by not showing that prompt until it's ready for more commands. Whatever you type into the Console will still be processed once R is available again, but you shouldn't rely on the currently running evaluation being successful. RStudio will also show a small stop sign above the Console while long-running calculations are keeping R busy, similar to the one shown in figure 1.8.

Each time you start up RStudio (or start R from the command line), you're working in a *session*. Within a session, the data (and functions) loaded into memory (the Workspace) are available to be queried and modified. This is a very different mindset to adjust to if you're coming from a spreadsheet-based environment where the data is ready and waiting when you open the program back up. Here, you're saving the *raw* data and steps to produce the output, which means your work is reproducible. Nothing is permanent until it's saved.

**NOTE** Packages and data that have been loaded are available only within a given session, so if you manually load a package in one session (for example using the `library()` function, covered fully in chapter 4), read in some data, or create anything, you'll

need to repeat those steps in any other sessions for which you want to use that package. By saving your *script* (a log of the commands you've entered), you can easily restart your session and catch back up to where you were. Scripts are merely plain-text commands saved to disk, and exist like any other file, independent of R sessions.

Commands that you want R to run when it starts a session should be placed in a file called `.Rprofile` in your home directory.<sup>8</sup> These commands could be calling `library()` on some packages you always intend to use, printing a message, or performing some common task.

---

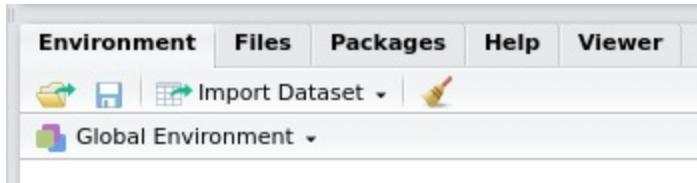
<sup>8</sup> This location depends on your operating system, but can be located from R itself as `sys.getenv('HOME')`.

If you try to exit R (either using the `q()` function or by closing the RStudio window), you'll likely be prompted to save your Workspace image. This allows you to keep everything you've defined (*assigned*, covered in the next chapter) within your Workspace in the session to reload later and resume where you left off. That may or may not be something you want to do, depending on what it is you're doing, but you should try to write your code in a way that will produce the same answers if you do have to restart without saving. That helps ensure your workflow is reproducible.

The setting can be changed if it becomes bothersome by clicking Tools > Global Options > General > Save Workspace to `.RData` on Exit. I recommend you set this to not saving by default so that you always start with a fresh session.

When you want or need to start from a fresh session with nothing defined and no extra packages loaded, you can close R or RStudio and reopen it. But if you're using RStudio, a faster option is to press Ctrl-Shift-F10, which will instruct RStudio to restart the session. In some versions this will also remove all the defined objects from the Environment. If it doesn't, you can click

the broom icon to clear the Workspace, as shown in [figure 1.9](#).

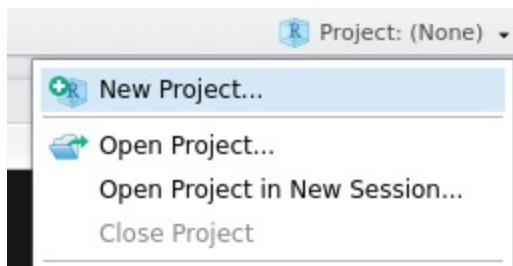


**Figure 1.9** Clean up after yourself.

It's a good idea to clean things occasionally to make sure that your script is reproducible — it's all too easy to shuffle a few lines around and convince yourself that you still get the outcome you expect when the objects already exist, but it breaks the sequential nature of the script and falls apart when an object is used before it's defined.

**TIP** You'll likely work on several different things as you discover how to work with R — sometimes more than one thing at a time. RStudio makes this easy to keep track of with *projects*. When you create an RStudio project, RStudio keeps track of which files you have open, where they're located, and even the positioning of the different panes specific to that project. Each project has its own R session, so you can run several independently for different contexts without worrying that your finance analysis will interfere with your mapping visualizations.

I highly recommend you start each distinct context of work in a new project. You can select projects from the top-right menu in RStudio, as shown in [figure 1.10](#).



**Figure 1.10** Selecting projects from the menu

Or you can create a new project with File > Project, or open an existing project with File > Open Project.

RStudio will try to warn you when you accidentally create code that doesn't make sense, possibly because it's in the wrong order, but it may not if a variable is defined at the current state in the session. A session isn't much use until you can do something within it, so let's move on to what you can tell R to do with some data.

## **1.4.2 BUILT-IN PACKAGES (DATA AND FUNCTIONS)**

R comes with a variety of packages preinstalled that are considered as the base version of R. It also bundles in a series of useful packages that will help you do most of what you need to do:

```
#> [1] "base"      "compiler"   "datasets"   "graphics"
"grDevices"
#> [6] "grid"       "methods"    "parallel"   "splines"
"stats"
#> [11] "stats4"    "tcltk"     "tools"     "utils"
```

We'll talk more about these, and the functions they contain, later. For the base package, you can view the list of functions by typing the following into the Console and pressing Enter:

```
help(package = "base")
```

That lists these functions (450+ of them in R 3.4.3) alphabetically. The stats package provides 300+ more functions. You certainly won't need *all* of these, but it should be clear that there's a lot of functionality to utilize in the standard R installation, even before you reach out and install additional packages. You don't need to do anything special to use the functions that these base packages provide; they're locked and loaded from the time you start up R.

In addition to useful functions, R comes preinstalled with example datasets that are useful for demonstrations. The most

commonly used ones are as follows:

- `mtcars` — Motor Trend Car Road Tests. This data, extracted from *Motor Trend* magazine in 1974, comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). This is the dataset we'll use in our examples most often. It's cliché, but clichés only get old if you've seen them too often, which in this case you probably haven't. If you need to refer to it quickly without an R session handy, a screen capture of the entire dataset is shown in the front matter.
- `iris` — A famous dataset that gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.
- `USArrests` — Violent Crime Rates by US state. This dataset contains statistics, in arrests per 100,000 residents, for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

If you have read other guides to using R, you'll likely have seen these mentioned. We'll use these for our examples also, but don't be afraid to test out your new skills using these and others. More example datasets (87 are listed in version 3.4.3) are named and briefly described in the help menu for the package `datasets`, which you can view by entering the following into the Console:

```
help(package = "datasets")
```

### **1.4.3 BUILT-IN DOCUMENTATION**

When correctly constructed, R packages, functions, and data come with some helpful documentation. This can be accessed from within an R session with the following syntax: a question mark followed by the name of the package, function, or dataset to query. For example, to learn more about the `mean()` function, you would enter the following into the Console:

```
?mean
```

If the name following ? is a package, function, or dataset that R is aware of, then the documentation for it will appear in the Help pane — otherwise, an error message will appear in the Console, such as

```
?nonExistentFunction
```

```
#> No documentation for 'nonExistentFunction' in specified
packages and
#> libraries: you could try '??nonExistentFunction'
```

To search for some text in the documentation (to look for something without knowing the name of it), use double question marks:

```
??mean
```

This search has some limitations, to say the least. Some better solutions are in progress, such as DataCamp's ([www.rdocumentation.org](http://www.rdocumentation.org)), but for now most problems are covered by reading the manual (the ? documentation) and/or searching the web.

RStudio will also help you write your R code by providing pop-up tips on syntax — hover over a function name to see the template and default arguments/options. You also get autocomplete for known functions and arguments: pause while typing the name of a function or data value to see potential completions, select these with the up and down arrow buttons, and press Tab to see potential completions. RStudio will also warn you when you've potentially misspelled a function or data value name. These options are configurable via Tools > Options > Code.

## **1.4.4 VIGNETTES**

A vastly underutilized (but invaluable when used correctly) feature of R is the ability to create an in-depth guide along with any package, known as a *vignette*. Vignettes can cover any topic,

but generally highlight the use of a few important functions that a package provides, similar to research papers or tutorials. At several pages long, they're much more insightful than the limited information provided in the help page, which is more of a brief usage guide.

These vignettes are built when you install a package, as in the following:

```
# The plot3D package produces 3D plots.
# Install it using
install.packages(pkgs = "plot3D")
```

Vignettes are available from the Console via the `vignette()` command, with the topic (name) of a vignette (and potentially the package it arises from with the argument `package`):

```
# Vignette: Fifty ways to draw a volcano using package
plot3D
vignette(topic = "volcano")
```

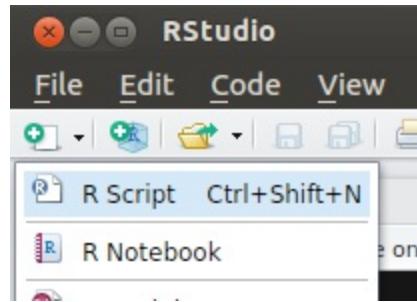
You can view all the currently available vignettes with this command:

```
browseVignettes(all = TRUE)
```

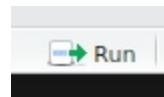
That will load a locally hosted web page with links to each of the available vignettes, arranged by the package they correspond to. They're generated as either PDF files, HTML (web) files, source files for these, or raw R source chunks. If you want to learn more about a package you've installed, this is a great place to check.

## ***1.5 Try it yourself***

If you haven't already done so, go ahead and open up RStudio and familiarize yourself with the program. Create a new folder in your home directory where you will save your work, and open a new R script, as shown in figure 1.11.



**Figure 1.11** Opening a new script



**Figure 1.12** Run button

Enter some commands into the script (try adding some numbers) and see them appear in the Console when you evaluate them. Press Ctrl-Enter (or Cmd-Enter) or click the Run button just above the Script pane, as shown in [figure 1.12](#).

Take note of how commands, results, errors, and warnings appear, how lines are evaluated, and how RStudio displays different pieces of code. Don't forget to save your script regularly!

## Terminology

- *Script* — A series of commands saved in a file (usually ending in .R or .r)
- *Value* — A piece of data
- *Variable* — A name to refer to a piece of data
- *Function* — Some code that interacts with data
- *Object* — A complex (or simple) structure that defines a variable or function
- *Package* — A collection of functions (and potentially data) that can be installed and used, and that extends the functionality of R
- *Vignette* — A long-form guide to using a package and its

functions, stored along with installed packages

## ***Summary***

- Data is everywhere.
- Handling of data comes with responsibility.
- Correctly structuring data makes it more accessible.
- Different types of data require different treatments.
- Visualizing data can uncover hidden patterns.
- Reproducible research means you can defend your results.
- R is used in many fields and has a long history.
- R interprets your commands rather than compiles into an executable.
- RStudio makes working with R code much smoother.
- R is extended by many packages.
- Reproducible research means creating a reproducible body of work by storing raw data and processing steps required to achieve a result, rather than merely storing the result.
- Each running instance of R represents a session. Assigned variables and loaded packages/functions are only available within a session and must be re-created if a new session is started.
- A visible prompt (>) in the Console means R is ready for your commands.
- You can write your commands in a script or one at a time in the Console.
- To exit the session, either close RStudio via the close button or enter the command `q()`.
- To find out more about a function, type the function name

preceded by a question mark, such as ?mean.

## 2

# Getting to know R data types

---

## This chapter covers

- Types of data you will likely encounter
- How R stores data using types
- How to assign values to variables

In any programming language, data is stored by the computer in memory using binary values. A *binary value* has only two possible states, and we can think of these as TRUE and FALSE, or more commonly, as 1 and 0. In computer memory, these represent the presence or absence of an electrical charge, so they are also ON and OFF.

We may see a 42 on the screen, but that's thanks to the computer translating what it has stored into what we expect to see. This can be extremely useful (better than seeing 00000000 00000000 00000000 00101010) but also somewhat dangerous, as we'll discuss later.

There are many different ways to store a piece of data in computer memory. By instructing the computer to treat data as a certain type, we can change how the data is manipulated.

## 2.1 Types of data

R is what's known as a *weakly typed* language, in that the *type* of

data is guessed or assumed rather than *declared* (or enforced). This can be very useful in terms of getting started because you don't need to worry about *setting* the data type whenever you're using data. But you'll also learn how to *change* the type later if you aren't getting what you want by default.

**NOTE** Some languages, such as C, are *strongly typed*, and in these languages the data type must be explicitly declared before it's used. This too has advantages, such as knowing that the data type can't be accidentally changed.

Several data types are available in the R language, and we'll go over the most common ones that you're likely to encounter. A quick summary of these is shown in table 2.1.

**Table 2.1** Data types

Data type	Examples
Numbers	1 3.14 -99
Text (strings)	"abc" "x" 'I said ''Hello'''
Categories (factors)	Months ("Jan", "Feb", "Mar") Colors ("red", "blue", "green") Countries ("Australia", "Japan", "USA")
Dates and times	2016-10-31 2000-01-01 00:00:01 UTC

Logicals	TRUE FALSE
Missing values	NA NaN

## 2.1.1 NUMBERS

The most common type of data is numbers in one form or another. Counts of birds, heights of trees, visitors to a website, temperatures, speeds, prices — all of these are numbers, but not all of them should be represented the same way. For some, it only makes sense to talk about whole numbers. The formal name for this type of number is *integer*. Integers are stored by the computer in a specific way. More general numbers, which might have a whole part and a fractional part, are sometimes called <sup>1</sup>*real* numbers. In R, these are called *numeric* values.

---

<sup>1</sup> For those more interested in computer science, these are double-precision values.

**DEFINITIONS** *Integer* means *whole* or *untouched*. Integers typically arise when you’re dealing with counting things that can’t be divided into smaller parts: 5 sheep, 3,000,000 people, 35 words. Integers have no *fractional* part to them.

*Real numbers* are also called *numerics* or *doubles*. These may have both an integer part and a fractional part, though the fractional part may be zero, in which case the value is equivalent to an integer. These arise from measurements that can take *any* value: 3.5 hours, 154.2 cm, 1.44 MB.

When you type something into the R Console and press Enter, R evaluates the expression and returns the value. Try it out by entering 3.2 into the Console and pressing Enter. You should see

the following:

```
3.2
#> [1] 3.2
```

Don't worry about the [1] for now — we'll come back to that. You may very well ask, "What type did I just use?" You didn't specify, so R took a reasonable guess. You can examine the storage type of something (as well as the general structure) using the built-in command `str()` (short for structure).

Try entering the following into the Console:

```
str(object = 3.2)
#> num 3.2
```

R reports that the *object* 3.2 is of type `num` (short for *numeric*) and has the value 3.2. Let's try that again with an *integer*:

```
str(object = 7)
#> num 7
```

R reports that this object is still a numeric type. Maybe you weren't specific enough:

```
str(object = 7.0)
#> num 7
```

Still numeric. So, what happened? By default, R will assume that a number entered with or without a decimal is of type numeric. To specify that you explicitly want an integer value, type an *L*<sup>2</sup> after the value:

---

<sup>2</sup> Why *L*? Well, *I* looks too much like a *1*, but the true reasoning behind the choice may have more to do with "long" numbers in the C language. *Literal* is a useful mnemonic for this.

```
str(object = 7L)
#> int 7
```

In each of these cases, the computer takes the value you give it (3.2 or 7L) and translates that into something it can store (an *object*), which can be represented by ones (1) and zeros (0).

There are various schemes for doing this, and they are typically different for numeric and integer types, but for now it's sufficient to know that those two types are different.

The `str()` function is very useful for inspecting the *structure* of an object, but when all you're interested in is the *type* of an object, a simpler function can help: the `typeof()` function. Let's try that with a value:

```
typeof(x = 7)
#> [1] "double"
```

The `typeof()` function reports slightly differently than `str()`, but *numeric*, *real*, and *double* all mean the same thing.<sup>3</sup>

---

<sup>3</sup> If you do a lot of high-level mathematics, you may also require the *complex* type such as  $3 + 2i$ , which `str()` and `typeof()` report as `cp1x` and `complex`, respectively.

Note that the *argument* names used in these two examples are different: `str()` has the argument `object`, whereas `typeof()` has the argument `x`. Every function has its own definitions of how its arguments are named, and there's not always rhyme or reason to these. Trying to use the wrong argument name leads to trouble. This

```
typeof(object = 7)
```

would lead to this:

```
#> Error: unused argument (object = 7)
```

Alternatively, trying with an explicit integer gives the following:

```
typeof(x = 7L)
#> [1] "integer"
```

Part of what makes this so confusing is that when R prints a value *equivalent* to an integer (`7L`, `7`, or `7.0`), it just prints the value without a decimal (`7`), so all of these look the same on the screen:

```
#> [1] 7
```

```
7.0
```

```
#> [1] 7
```

```
7L
```

```
#> [1] 7
```

One important thing to keep in mind is that R (or pretty much any programming language) won't keep track of your measurement *units* for you, it will only store the number *values*.

**NOTE** Measurements are composed of two parts: a value and a scale. We may tend to abbreviate this when we talk ("I was driving at 60"), but this only makes sense if we assume everyone is familiar with the *units* (was that kilometers per hour or miles per hour?). You may store population counts on the scale of "millions of people," but the *value* you use may be only 318.9.

A helpful way to ensure that calculations are consistent with respect to units is to name your variable according to the units it refers to (I'll cover this shortly).

## On keeping track of units

In September 1999, the Mars Climate Orbiter probe began its descent into the Martian atmosphere. The optimal altitude for the required maneuvers was 226 km (140 miles), but calculations performed later showed it was actually on a path that would place it a mere 57 km (35 miles) from the surface. The probe was destroyed by the atmospheric stress. The team at Lockheed Martin had made their calculations for the thrusters in pound-seconds, but the NASA trajectory calculating software expected Newton-seconds. Units are important!

There are other ways to input numbers. If you need to input very large numbers such as millions, you can use something similar to *scientific notation*, where you specify an *exponent* in the power of tens with an e or E. The number 1,000,000 has six zeroes, so 5 million can be entered like this:

```
5e6      ①
#> [1] 5e+06
```

①

Or 5E6

The part of the number in front of e or E can have decimal places if needed, so 1,200,000 (“one point two million”) can be entered as follows:

```
1.2e6
#> [1] 1200000
```

If you’re more specific about the decimal places than the power of 10 exponent, this will carry through:

```
3.14159e3
#> [1] 3141.59
```

To ensure that this value is an integer, you can add the L suffix to the *end* of the input:

```
typeof(x = 1.2e6L)
#> [1] "integer"
```

R will make the *entire* value an integer. Note that there is no way to use a numeric value in the power of ten exponent; it’s always assumed to be an integer, and entering something with a decimal generates an error.

Although it may be tempting to enter large numbers with thousands-separating commas or periods (as in 1,200,000), R won’t know what to do:

```
1,200,000
```

```
#> Error: unexpected ',', in "1"
```

This is particularly important when reading in numeric data from an external, real-world source. You'll see how to deal with this situation in chapter 6.

## 2.1.2 TEXT (STRINGS)

Numbers are the most common data type, but the next most common would be strings. *Strings* are groups of characters, letters, or any form of text. R treats strings differently because they *are* different. They represent an entirely different type of data, but a much richer one.

Strings are entered into R using either double quotes ("apple") or single quotes ('apple'), and both are treated the same. You can examine the structure of this data:

```
str(object = "apple")
#> chr "apple"
```

R reports that the object "apple" is of type `char` (short for *character*), and this is how strings, composed of zero or more characters, are stored. R doesn't particularly mind what's inside the quotes, so for example, an empty string is perfectly fine:

```
str(object = "")
#> chr ""
```

If you're likely to want to include either single or double quotes themselves in a string, it's a good idea to use the other type of quotes for defining the string. I recommend using double quotes by default ("x") and single quotes when you want to include quoted text ('He said "Hello"').<sup>4</sup> Alternatively, you can specify that the quotes inside your string are not to be interpreted, using a *slash* (\") to *escape* the character, so they won't end the string:

---

<sup>4</sup> Many editors will automatically insert the paired double quote for you when you enter ".

---

```
"The sign said, \"Walk\"."
```

```
#> [1] "The sign said, \"Walk\"."
```

The mixture approach, such as

```
"The sign said, 'Walk'."  
#> [1] "The sign said, 'Walk'."
```

means you can use apostrophes in your text without issue:

```
"That's John's father's name."  
#> [1] "That's John's father's name."
```

With the alternative arrangement

```
'The sign said, "Walk".'  
#> [1] "The sign said, \"Walk\"."
```

the double quotes have been *escaped* (preceded by \) for you.

## Escape character

There are several *special characters* in R that have a particular meaning inside strings. You can enter these by immediately preceding them with a backslash (\).

The ones you're most likely to encounter are

- \' — Single quote
- \" — Double quote
- \n — Newline
- \\ — Backslash itself

When a string is merely printed, the escape character is also printed for clarity. The function `cat()` (short for *concatenate*, another word for *join*) interprets the special characters when it outputs something to the screen:

```
cat('\\'Special characters\\ndo \\'special things\\'')  
#> 'Special characters'  
#> do 'special things'
```

Notice how the string is both defined with, and uses, single quotes, and the explicit line break (using `\n`). Take caution when using a backslash in strings in case you unintentionally create a special character. If you require a backslash, it's best to escape it (`\\\`).

Behind the scenes, something is happening similar to when you store a number. A computer has no way to literally store the letter *a*, so it converts it to ones and zeros using an *encoding* scheme (basically a look-up table of values and their corresponding representations on the screen). There are even more ways to do this, and unfortunately there's no perfect consensus on how, so occasionally computers get confused about how to translate between the stored version and the version you see on the screen.

If you've ever received a garbled email that looks something like “â € ™”, it means the encoding has somehow been incorrectly set. In this case, an apostrophe (') is encoded using a scheme called UTF-8 as three *hexadecimal* (base 16, using 0–9 and A–F) values: `0xE2 0x80 0x99` (alternatively written as `e28099`); but those three values in the Windows-1252 encoding look-up table correspond to the symbols â, €, and ™, so if the computer gets confused about which scheme to use, the text appears different than what was expected.

You don't need to worry about how the computer does this, for the most part. R will happily work with the text you type in.

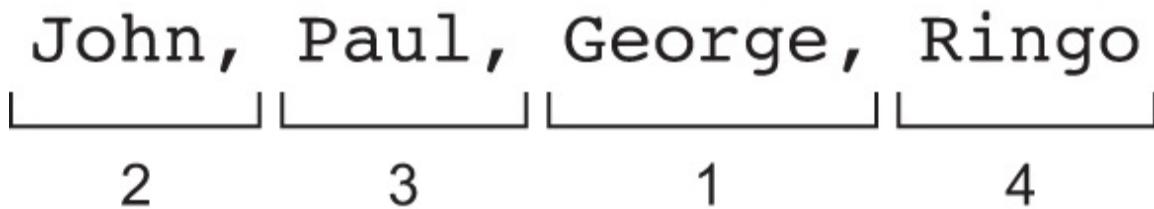
### **2.1.3 CATEGORIES (FACTORS)**

Sometimes you want to work with categories of things, with *categorical data* as opposed to *continuous data*, which could be any value within some range. When working with categories, there are a limited number of choices from which your values might be taken, and it can be useful to let R know that this is the case:

- *Categorical data* — When you have a limited set of things you are describing, you can consider them to be in their own distinct little boxes, one for each category. There exists some distinction between each of them that makes the categories meaningful. Country names, for example, are distinct, so these can form categories.
- *Continuous data* — When you have a range of values without distinct differences between them, placing those values in meaningful boxes becomes tricky. Rather than creating smaller and smaller boxes, you can consider the entire range of values together. Consider the range of heights among a sample of adults; depending on how precisely you are measuring, this could be any value between the shortest and tallest heights.

When the values are numbers (numeric or integer) but only certain values are to be considered, we call these *discrete values*. These might be the number of cylinders in an engine (4, 6, 8) or television screen dimension measurements (30, 36, 40, 52, 60). They don't need to be integers, they just need to have specific values (3.2, 9.4, 7, 129.4). R will store their labels as `char` (text) either way.

Alternatively, you might have categories that are described by text, such as names ("John", "Paul", "George", "Ringo") or places ("Los Angeles", "New York", "Chicago", "Seattle"). When you have data like this and want R to know that there are only certain values available, you can convert the data to the type `factor`, in which case R creates a look-up table of the names for your categories (either the numbers, text, or whatever type you have) and the category *index* (number, with some ordering), which you can see in figure 2.1.



**Figure 2.1 Beatles category index**

This allows R to only have to use the category *values* once, and it can resort to the index whenever it needs to do something with it. This was once very useful in terms of memory saving, but that aspect has become less important as memory has become so easily available.

The ordering is typically alphabetical (by default), and you can see how R has chosen to do this by using `str()`. Enter the following into the Console, either line by line or by copying all of it and pasting it all at once:

```
str(object =  
  factor(x =  
    c( "John", "Paul", "George", "Ringo"  
  ))  
)  
#> Factor w/ 4 levels "George", "John", ... : 2 3 1 4
```

①

`str(object)` asks for the structure of object.

②

`factor()` tells R to create a factor object from the values in the argument `x`.

③

`c()` creates a concatenated (joined) group, which you'll see in chapter 5.

In this case, the categories (the names) are ordered alphabetically, so "George" receives index 1, "John" receives index 2, "Paul" receives index 3, and "Ringo" receives index 4. This can be seen in the `str()` output, which lists the `factor` levels and then describes the data by these index values: 2 3 1 4.

**CAUTION** Be careful not to confuse the index values with the ordering: in the preceding example, the values "John", "Paul",

"George", "Ringo" were assigned the alphabetical indices 2 3 1 4. This is not a reordering; the data still appears in the sequence specified, but now the names have been replaced by a number. You can return to the original names by replacing the other way, so 1 becomes "George", and so on.

Factors can be very useful when you're dealing with categories because they *abstract* away the category labels into their index. Sometimes, you want to preserve some information about those categories — the fact that they have a natural order. One example is when categories are groups or ranges of values: "small", "medium", "large". In this case, keep in mind that the "small" category represents things that are smaller than those in the "medium" category, which themselves are smaller than those in the "large" category:

```
str(object = factor(x = c("medium", "small", "large")))
#> Factor w/ 3 levels "large","medium",...: 2 3 1
```

Note that the order (alphabetical) is the opposite of what you probably want.

In this case, you can tell R that this is an *ordered factor*, in which case it will use the ordering specified in the `levels` argument:

```
sizes <- factor(x      = c("medium", "small", "large"),
                 levels = c("small", "medium", "large"),
                 ordered = TRUE)
str(object = sizes)
#> Ord.factor w/ 3 levels "small"<"medium"<...: 2 1 3
```

This becomes important when you want outputs to display in a particular order:

```
table(sizes)
#> sizes
#>   small medium  large
#>     1       1       1
```

Factors can be a source of great frustration when they show up

unexpectedly, so it's a good idea to know when and where R will try to help you out a bit too much and use them.

## Dial m for Male. Or M. Or male.

Even in many professional settings, data is often entered by hand. This can lead to conventions not being followed, and different names for things being used when a consistent value should be. One common example is entering male and female identifiers. When R sees a text column that it needs to convert to a factor, it generates the full list of options and assigns the index of each occurrence. If most of your values in a Sex column are `male` or `female` but the occasional `m`, `M`, `maIe`, or `Male` has slipped through, R will gladly — and silently — create this aberration as a new factor level.

The best way to not get caught by this is to inspect the levels created whenever this might be an issue. The `levels()` function returns the unique levels of a factor variable:

```
levels(x = sizes)
#> [1] "small" "medium" "large"
```

This is slightly different than the behavior in Excel, which is known to ignore upper/lowercase when filtering values.

### **2.1.4 DATES AND TIMES**

When dealing with data that has been collected at (or describes) a particular date and time, you need to deal with these structures. Numbers aren't sufficient, and strings don't have any rules. R has several special types for dealing with dates and times.

When you have dates without times, the `Date` type is most useful, but if you also have times associated with those dates, the

`POSIXct` type allows that functionality. Figure 2.2 shows an example.

```
"2017-02-03" —> Date
"2017-02-03 23:16:59 CST" —> POSIXct
```

**Figure 2.2 Date types**

You can examine examples of each of these structures by asking R to produce the current time or date as the system understands it to be using two built-in functions, `sys.time()` and `sys.Date()`. The former produces, as you may well guess, the current time, and the latter produces the date:

```
str(object = Sys.time())
#> POSIXct[1:1], format: "2018-01-23 21:38:47"
```

Here R tells you that the time is stored as type `POSIXct` and gives an example of the format. You can also be less specific and request the date:

```
str(object = Sys.Date())
#> Date[1:1], format: "2018-01-23"
```

Have you ever read the date as someone's written it and wondered what order they intended? Is `04-08-16` the 4th of August 2016, the 8th of April 1916, or the 16th of August 2004? For the purposes of consistency and reproducibility, it's important to choose an order and stick to it. I highly recommend the ISO-8601 standard method of writing dates in the format of `YYYY-MM-DD` for year-month-day, regardless of what is more common where you happen to live. This makes dates easy to sort and is generally easier to work with.

R requires that the input format be specified using percent signs and certain codes, which are described in another help file (`?strptime`). R uses the default format of `"%Y-%m-%d %H:%M:%S"` with the codes specified in table 2.2. Note also that the capitalization is important: `%m` represents *month*, while `%M`

represents *minute*.

**Table 2.2 Date and time input format codes**<sup>5</sup>

Code	Meaning	Range
%Y	Year (including century)	1–9999
%m	Month	01–12
%d	Day	01–31
%H	Hour	00–23
%M	Minute	00–59
%S	Second	00–59

---

<sup>5</sup> For more, see `?strptime`.

There are many additional codes, for example to specify a month abbreviation such as “Feb” (%b), and these are listed in the help menu for the function `strptime`, obtained by evaluating `?strptime`.

Working with dates in this format is made easier with additional packages, but the most basic way to extract a part of a date or time object (say, the year) is to change the printed format to include the structure you want to extract. To extract the current year from a call to `sys.time()`, you can achieve this with the `format()` function, which formats its input into a character string. It takes the same format codes as the date and time functions, so you can extract the year (as YYYY with %Y):

```
format(x = Sys.time(), format = "%Y")
#> [1] "2018"
```

An additional complication is the time zone that a `Date` or `POSIXct` value corresponds to. R is certainly able to handle this, but as a first precaution it’s advisable to ensure that all your data corresponds to a single time zone.

## 2.1.5 LOGICALS

The values `TRUE` and `FALSE` can be extremely useful, and as such these are reserved names in R, which means they can't be reassigned. R reports these as type `logi` (for *logical*, sometimes referred to as *boolean*):

```
str(object = TRUE)
#> logi TRUE
```

These become highly useful when you want to execute code *conditionally* based on some other calculation, which I cover in chapter 8.

**NOTE** For historical reasons, the values `T` and `F` also exist in R and have the values `TRUE` and `FALSE`, respectively. Unlike their full-named counterparts, however, these aren't reserved names, so they *can* be assigned (I highly recommend against using either of these as variable names for this reason). If someone felt evil enough, they could cause a great deal of confusion by setting these to their opposite value deep within some code and then testing these values further down.

`TRUE` and `FALSE` are frequently used with `binary` data — which can only be in either of two states. Alive or dead, up or down, on or off, yes or no, or literally true or false. By storing this type of data as `logical`, you ensure that if something has a value, it can only take one of these values.

Settings are also frequently stored with this type. You can turn on (or off) the printing of quotation marks when you `print()` results:

```
print(x = "abc")
#> [1] "abc"

print(x = "abc", quote = FALSE)
#> [1] abc
```

Sometimes, although you want your data to be stored eventually, you don't know what value it has. In that case, there's a third option that fits neither of these labels: you can explicitly specify that some data isn't there; it's missing.

## 2.1.6 MISSING VALUES

R has several ways of denoting data that is missing. The most common of these is `NA` (not available), which represents data that is skipped over or not present. Although this value may be not present, R can still keep track of what *type* it was supposed to be. There are a few flavors of `NA`, even though they're still missing data: `NA_real_`, `NA_integer_`, and `NA_character_` are the ones you're most likely to encounter. By default, `NA` is a logical type:

```
typeof(x = NA)
#> [1] "logical"
```

### Don't ignore what's missing

Missing values can be extremely important information, so take care about how you use them. It's not at all uncommon to see people producing spreadsheets with a mixture of empty cells, sentinel values (implausible values that signal something else, such as -99), zeroes, and text representations of missingness such as "N/A" or "...".

Missingness might represent the fact that an observation or measurement wasn't taken or performed, rather than that it produced a value of 0. That distinction becomes very important when 0 is a meaningful value for the observation/measurement, such as a count of something. Many of R's statistical functions have special options for dealing with missing values that are coded as `NA`, typically the TRUE/FALSE option `na.rm` which instructs that a function should remove `NA` values before calculating.

`NULL` represents part of a structure that's not just missing but nonexistent entirely;<sup>6</sup> the underlying structure of a `NULL` object is inherently different. `NA` and `NULL` differ in that `NA` represents a value that could have existed but doesn't, whereas `NULL` represents the lack of even the structure in which the value could have existed. `NaN` (not a number) refers to values that are undefined (such as  $0/0$ ), though these are rarely encountered.

---

<sup>6</sup> Technically, there is only a `singlenull` object in any R session, and any instances of variables given this value point to that.

---

In addition to these, the value `Inf` refers to the “value” of infinity (such as produced by evaluating  $1/0$ ), which is larger than any other number. You can also refer to `-Inf`, which is smaller than any number. This isn't really a missing number, but at the same time it isn't a number you can do much with. But it can be useful when comparing values (which you'll see in chapter 3) as `Inf` is larger than any other value.

## 2.2 Storing values (assigning)

Up until now you've only been asking R for the *structure* of data, and you haven't particularly done anything with it. The function `str()` merely prints details about the structure of something to the screen — it doesn't *assign* (name and keep) any values for later use.

In order to do something with data, you need to tell R what to call it so that you can refer to it in your code. In programming in general, we typically have *variables* (things that may vary) and *values* (data). You've already seen that different data values can have different *types*, but you haven't told R to store any of them yet. Next, you'll create some variables in which to store data values.

### 2.2.1 NAMING DATA (VARIABLES)

If you have the values 4 and 8 and want to do something with them, you can use the values literally (say, add them together as

4 + 8). You may be familiar with this if you frequently use Excel or some other spreadsheet software; data values are stored in cells (groups of which you can choose to name), and you tell the program which values to combine in some calculation by selecting the cells with the mouse or keyboard. Alternatively, you can choose to refer to cells by their grid reference (for example, A1). An example of this is shown in figure 2.3.

	A	B	C
1	4	8	=A1+B1
2	5	9	
3	6	10	

**Figure 2.3 Spreadsheet formula to add two cell values**

When the cell formula is evaluated, the result takes its place, as shown in figure 2.4.

	A	B	C
1	4	8	12
2	5	9	
3	6	10	

**Figure 2.4 Spreadsheet formula result**

Similar to the A1 and B1 references, you can store values in *variables* (things that may vary, also called *objects*) and abstract away the values. In R, assigning of values to variables takes the following form:

```
variable <- value
```

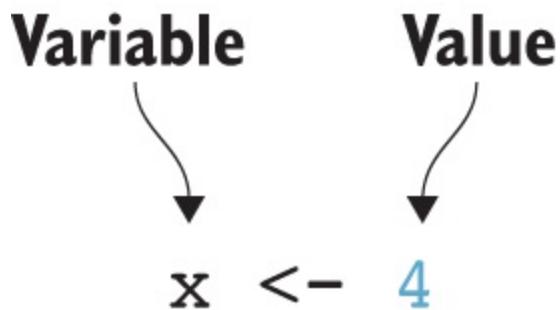
The *assignment operator* <- can be thought of as storing the value/thing on the right-hand side into the name/thing on the left-hand side.

Try typing the following into the R Console and then pressing Enter:

```
x <- 4
```

A diagram of the components involved in assignment is shown in figure 2.5. You could just as easily use the equals sign to achieve

this, with  $x = 4$ , but I recommend you use  $<-$  for reasons that will become clear later. It may seem odd that multiple characters are required to type  $<-$  but you'll get used to it. If you're using RStudio (and I hope you are), there's a shortcut to entering this, with default keys Alt+- (Alt and dash) on PC, Option+- (Option and dash) on Mac.



**Figure 2.5 Components of an assignment operation.**

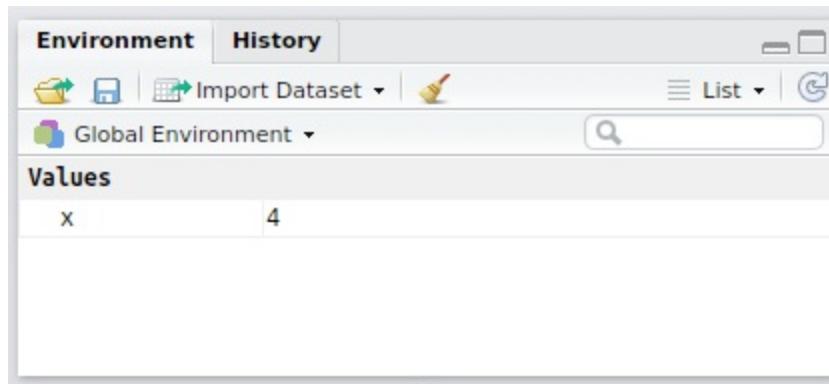
**NOTE** Back when the S language was developed, the APL keyboard was popular. It featured a  $\leftarrow$  key, so this was even easier to input. An example is shown in figure 2.6.



**Figure 2.6 APL keyboard, adapted from Wm313 Creative Commons Wikipedia (<http://mng.bz/mzKW>)**

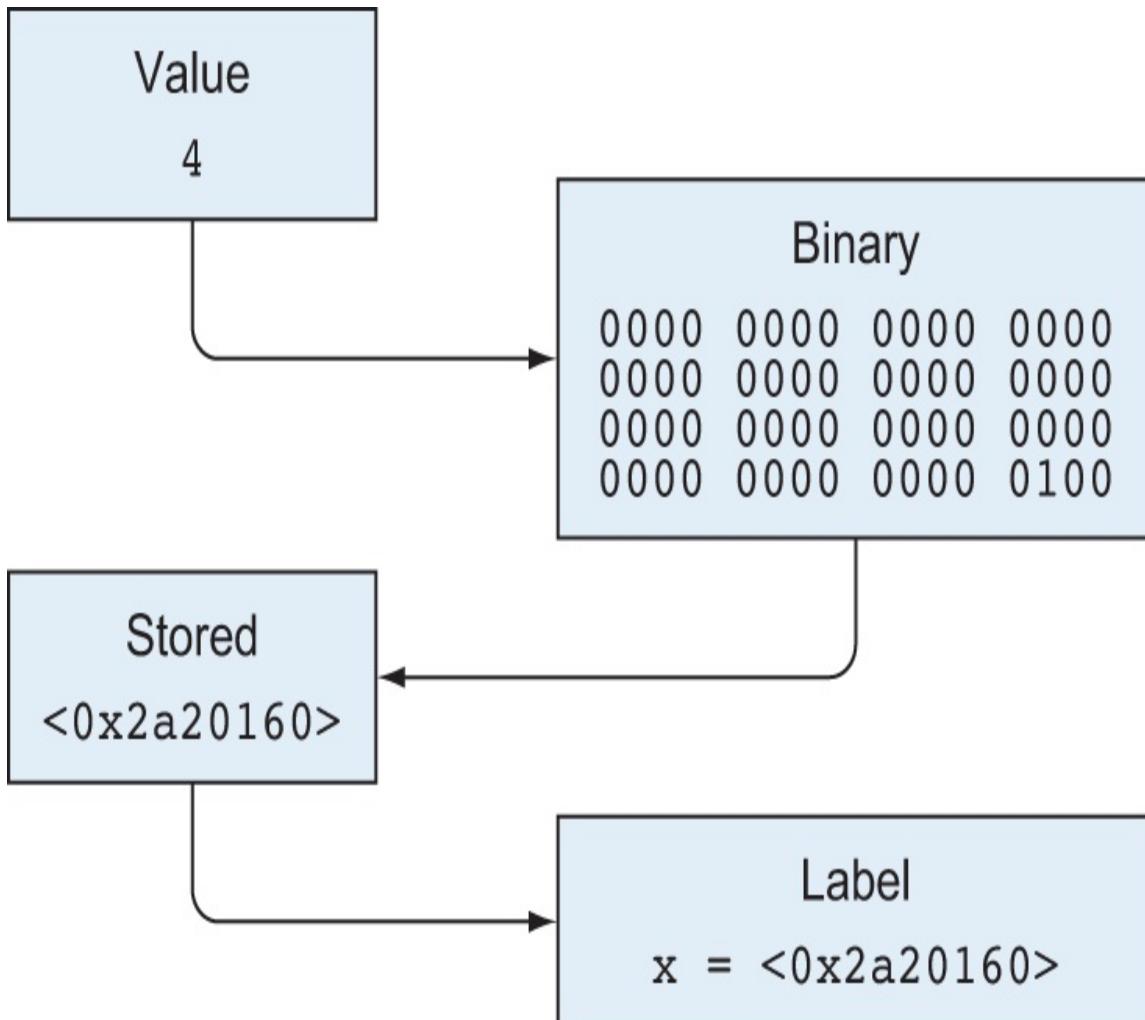
You'll notice that the Environment tab of the Workspace pane now lists  $x$  under Values and shows the number 4 next to it, as shown in figure 2.7. Your Environment listing may show more

variables, depending on what else you've assigned in your session so far.



**Figure 2.7 Environment pane showing the value 4 assigned to the variable x**

Behind the scenes, when you pressed Enter, R took the entire expression you entered (`x <- 4`) and evaluated it. Because you told R to *assign* the value 4 to the variable x, R converted the value 4 to binary and placed that in the computer's memory. R then gives you a reference to that place in the computer's memory and labels it x. A diagram of this process is shown in [figure 2.8](#). Nothing else appeared in the Console because the action of assigning a value doesn't *return* anything (I'll cover this more in chapter 4).



**Figure 2.8** R's process of converting values

## Forcing R to print with parentheses

If you really want to force the printing of a value during assignment, you can do so by surrounding a non-returning (or any) expression with parentheses. Normally an assignment prints nothing:

```
no_return <- 7
```

But you can force it to if you start the expression with ( and end with ):

```
(return_me <- 7)
```

```
#> [1] 7
```

This is particularly handy to show what you've just assigned, either for testing or demonstration. This should be restricted to single lines<sup>7</sup>, though, because it won't work on multiline expressions.

---

<sup>7</sup> You may be able to also surround an expression in curly braces and then parentheses to allow this, but at that point you should really wonder why you need to do so.

This is overly simplified, of course. Technically speaking, in R, names have objects rather than the other way around. This means that R can be quite memory efficient since it doesn't create a copy of anything it doesn't need to.

**CAUTION** Variables that begin with a period (`.length`) are considered *hidden* and don't appear in the Environment tab of the Workspace. They otherwise behave exactly like any other variable: they can be printed and manipulated. An example is the `.Last.value` variable, which exists from the moment you load up R (with the value `TRUE`) — this contains the output value of the last statement executed (handy if you forgot to assign it to something). There are very few reasons you'll want to use this feature (dot-prefixed hidden variables) on purpose at the moment, so for now, avoid creating variable names with this pattern. You can request `.Last.value` to be visible in the Environment tab via Tools > Global Options > General > Show.Last.value in Environment Listing. This option is sometimes under an Advanced subtab within General.

You can retrieve the value assigned to the variable `x` by asking R to print the value of `x`:

```
print(x = x)
#> [1] 4
```

Here's a useful shortcut — if your entire expression is just a variable, R will assume you mean to `print()` it, so the following works just the same:

```
x  
#> [1] 4
```

Now, about that `[1]`: it's important to know that in R, there's no such thing as a single value. Every value is a *vector* of values (I'll cover these properly in the next chapter, but think of these as collections of values of the same type).<sup>8</sup> Whenever R prints a vector, it allows for the case where the vector contains more than one value. To make this easier on the eye, it labels the first value appearing on the line by its position in the collection. For collections (vectors) with just a single value, this might appear strange, but it makes more sense when variables contain more values — for example, if you print the column names of the `mtcars` dataset:

---

<sup>8</sup> In technical terms, R has no *scalar* values.

```
colnames(x = mtcars)  
#> [1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec"  
"vs"    "am"    "gear"  
#> [11] "carb"
```

You can assign another value to another variable:

```
y <- 8
```

You can create as many variables as you need, as long as they fit within the available memory. All R objects are stored in the computer's RAM, so this limits the size of these objects.<sup>9</sup>

---

<sup>9</sup> You would need to create a *lot* of large variables to exhaust even a modest amount of RAM on a modern computer.

There are few restrictions for what you can name your variables, but R will complain if you try to break them. Variables should start with a letter, not a number. Trying to create the variable `2b` will generate an error:

```
2b <- 7
#> Error: unexpected symbol in "2b"
```

Variable names can start with a dot (.) as long as it's not immediately followed by a number, although you may want to avoid doing so (these are “hidden” variables, but otherwise work as normal). The rest of the variable name can consist of letters (upper- and lowercase) and numbers, but not punctuation (except . or \_) or other symbols (except the dot, though again, preferably not).

If you absolutely need to bypass these restrictions, you can surround your variable name in quotes or backticks when assigning, and refer to it with backticks when using it. There are very few situations in which this will be a good idea, but it's good to know it's possible. For example, here's a variable name that starts with a number:

```
"2b" <- 7
7 + `2b`
#> [1] 14
```

And here's one that includes spaces:

```
`to be or not to be` <- 2
7 + `to be or not to be`
#> [1] 9
```

Trying to access these with quotes won't work (you just get the character string):

```
"2b"
#> [1] "2b"
```

They must be referenced with backticks:

```
`2b`
#> [1] 7
```

This will become important later when you read in external data.

There are also certain *reserved words* that you can't use as variable names. Some are reserved for built-in functions or keywords, most of which I'll cover later: if, else, repeat, while,

function, for, in, next, and break. Others are reserved for particular values: TRUE, FALSE, NULL, Inf, NaN, NA, NA\_integer\_, NA\_real\_, NA\_complex\_, and NA\_character\_. I've covered what each of these means, so hopefully you can see why you can't create a variable with one of those names.

What you *can* do, though you may want to take care with it, is *overwrite* the built-in names of variables and functions. By default, the value pi is available ( $\pi = 3.141593$ ). If you were translating an equation into code and wanted to enter a variable referring to the  $i$  th value of p,  $p_i$ , you might accidentally call it pi and in doing so change the default value, causing all sorts of trouble when you next use it or call a function you've written that expects the value to still be the default.

The default value can still be accessed by specifying the package in which it's defined, separated by two colons (::). In the case of pi, this is the base package:

```
pi <- 3L      ①
base::pi      ②
#> [1] 3.141593
```

①

**Redefining pi to be equal to exactly 3**

②

**The default correct value is still available.**

This is also an issue for functions, with the same solution: specify the package in which it's defined to use that definition. We'll return to this later.

The next section covers how to do things to variables in more detail, but for now let's see what happens if you add the variables x and y in the same way as regular numbers:

```
x + y
#> [1] 12
```

That's what you get when you add these numbers explicitly. Note that because the expression produces just a number (no assignment), the value is printed. I'll cover how to add and subtract values in more depth in section 3.1.

R has no problems with overwriting these values, and it doesn't mind what data you overwrite them with:<sup>10</sup>

---

<sup>10</sup> This is where the distinction of *weakly typed* becomes important. In a *strongly typed* language you wouldn't be able to arbitrarily change the type of a variable.

```
y <- "banana"
y
#> [1] "banana"
```

R is *case sensitive*, which means it treats a and A as distinct names. You can have a variable named `myVariable` and another named `MYvariable` and another named `myVARIABLE`, and R will hold the value assigned to each independently.

There are only two hard things in Computer Science: cache invalidation and naming things.

—Phil Karlton, Principal Curmudgeon, Netscape Communications Corporation

I said earlier that R won't keep track of units, so it's a good idea to name variables in a way that makes logical sense, is meaningful, and will help you remember what they represent. Variables `x` and `y` are fine for playing around with values, but they aren't particularly meaningful if your data represents speeds, where you may want to use something like `speed_kmph` for speeds in kilometers per hour. Underscores (`_`) are allowed in variable names, but whether you use them is up to you. Some programmers prefer to name variables in this way (sometimes referred to as *snake\_case*), others prefer *CamelCase*. The use of periods (dots, `.`) to separate words is discouraged for somewhat more advanced reasons.<sup>11</sup>

---

<sup>11</sup> This syntax is already used within R to denote functions acting on a

specific class, such as `print.Date()`.

**TIP** Take care when naming variables. Make them meaningful and concise. Six months from now, will you remember what `data_17` corresponds to? Tomorrow, will you remember that `newdata` was updated twice?

## 2.2.2 UNCHANGING DATA

If you're familiar with working with data in a spreadsheet program (such as Excel), you may expect variables to behave in a way that they won't. Automatic recalculation is a very useful feature of spreadsheet programs, but it's not how R behaves.

If you assign two variables and then add them, you can save that result to another variable:

```
a1 <- 4
a2 <- 8
sum_of_a1_and_a2 <- a1 + a2
```

This holds the value you expect:

```
print(x = sum_of_a1_and_a2)
#> [1] 12
```

It's also the same result you'd get from using, say, `A1+A2` in a spreadsheet, as shown in figure 2.9.

	A
1	4
2	8
3	=A1+A2

	A
1	4
2	8
3	12

**Figure 2.9** Adding two values in a spreadsheet

Now, if you *change* one of these values in R

```
a2 <- 7
```

this has *no impact* on the value of the variable you created to hold the sum earlier, or on any other variables in the Workspace:

```
print(x = sum_of_a1_and_a2)
#> [1] 12
```

By contrast, a spreadsheet would most likely recalculate the sum. Note also that the value of `sum_of_a1_and_a2` shown in the Environment pane hasn't changed when you reassign `a1` or `a2`.

Once the sum is calculated, and that value stored in a variable, the connection to the original values is lost. This makes things *reliable* because you know for sure what value a variable will have at any point in your calculation by following the steps that lead to it, whereas a spreadsheet depends much more on its current overall *state*.

### **2.2.3 THE ASSIGNMENT OPERATORS (<- VS. =)**

If you've read some R code already, you've possibly seen that both `<-` and `=` are used to assign values to variables, and this tends to cause some confusion. Technically, R will accept either one when assigning variables, so in that respect it comes down to a matter of style (I still highly recommend assigning with `<-`). The big difference comes when using functions that take *arguments* — there you should only use `=` to specify what the value of the argument is. For example, when you inspect the `mtcars` data, you can specify a string with which to indent the output:

```
str(object = mtcars, indent.str = "INDENT>> ")
#> 'data.frame':      32 obs. of  11 variables:
#> INDENT>> $ mpg  : num  21 21 22.8 21.4 18.7 18.1 14.3
#> 24.4 22.8 19.2 ...
#> INDENT>> $ cyl  : num  6 6 4 6 8 6 8 4 4 6 ...
#> INDENT>> $ disp: num  160 160 108 258 360 ...
#> INDENT>> $ hp   : num  110 110 93 110 175 105 245 62 95
#> 123 ...
#> INDENT>> $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21
#> 3.69 3.92 3.92 ...
```

```
#> INDENT>> $ wt    : num  2.62 2.88 2.32 3.21 3.44 ...
#> INDENT>> $ qsec   : num 16.5 17 18.6 19.4 17 ...
#> INDENT>> $ vs     : num 0 0 1 1 0 1 0 1 1 1 ...
#> INDENT>> $ am     : num 1 1 1 0 0 0 0 0 0 0 ...
#> INDENT>> $ carb   : num 4 4 1 1 2 1 4 2 2 4 ...
```

If you used `<-` instead of `=` for either argument, then R would treat that as creating a new variable object or `indent.str` with value `mtcars` or `"INDENT>> "` respectively, which isn't what you want.

Play with assigning some variables, as shown in the following listing, and see if you always get the results you expect.

### [Listing 2.1](#) Try it yourself: assigning to variables

```
score <- 4.8
score
#> [1] 4.8
str(object = score)
#> num 4.8
fruit <- "banana"
fruit
#> [1] "banana"
str(object = fruit)
#> chr "banana"
```

Note that you didn't need to tell R that one of the variables was a number and one was a string — it figured that out itself. It's good practice (and easier to read) to make your `<-` line up vertically when defining several variables

```
first_name <- "John"
last_name  <- "Smith"
top_points <- 23
```

but only if this can be achieved without adding too many spaces (exactly how many is too many is up to you).

**Watch this space!**

An extra space can make a big difference to the syntax.  
Compare this

```
a <- "x"
```

with this:

```
a < - 3      ①
#> [1] FALSE
```

①

**< is the less-than operator.**

The first case assigns the value "x" to the variable a (which returns nothing). The second case, with a wayward space, compares a (which now holds the value "x") to the value -3, which returns FALSE (I'll explain why that works at all when we look at comparing values in section 3.4).

Now that you know how to provide some data to R, what if you want to explicitly tell R that data should be of a specific type, or convert data to a different type?

## 2.3 Specifying the data type

By default, R will assume the most general type for data as you assign it to a variable. You've seen that a value that only *looks* like an integer will still be stored as a numeric type, just in case

```
myInt <- 3
str(object = myInt)      ①
#> num 3
```

①

**This is a numeric type.**

unless you specifically tell R otherwise:

```
myInt <- 3L
```

```
str(object = myInt) ①
#> int 3
```

①

**Now it's an integer.**

Sometimes, though, data isn't of the type that you want it to be. So, how do you convert it? R has a group of functions that start with `as.` that attempt to convert (*coerce*) any input data into another type. The important such functions at this point are as follows:

- `as.integer()`
- `as.numeric()`
- `as.character()`
- `as.logical()`
- `as.POSIXct()` and `as.POSIXlt()`
- `as.Date()`

These take whatever data you have and convert it to a specific type. So, if you have a number, but you want R to treat it like text (character/string), you can use `as.character()`:

```
numAsString <- as.character(x = 1234)
str(object = numAsString)
#> chr "1234"
```

You can force this when entering the data by placing quotes around the number, making it a string:

```
numAsString <- "1234"
```

But sometimes you want to convert a value already stored in a variable:

```
num <- 1234
numAsString <- as.character(x = num)
str(object = numAsString)
#> chr "1234"
```

Again it becomes difficult (impossible) to distinguish between numeric and integer types once converted to character strings:

```
as.character(x = 7)
#> [1] "7"
```

```
as.character(x = 7L)
#> [1] "7"
```

More common perhaps would be to go the other way — to treat a character string as a number. Several functions will do that by default, but often you'll need to do the conversion yourself. If you have some numeric data that has been imported as character strings, you can convert these back to numbers:

```
stringAsNum <- as.numeric(x = "3.14159")
str(object = stringAsNum) ①
#> num 3.14
```

①

**Now numeric**

Don't worry that the `str()` function reports fewer decimal places. The entire value is still there with as many digits as you provided:

```
print(x = stringAsNum)
#> [1] 3.14159
```

This becomes slightly more difficult when dealing with dates. Sometimes (or perhaps more often than not) dates won't be in the default format of "%Y-%m-%d", in which case you'll need to use the optional `format` argument to specify this according to the codes in [table 2.2](#).

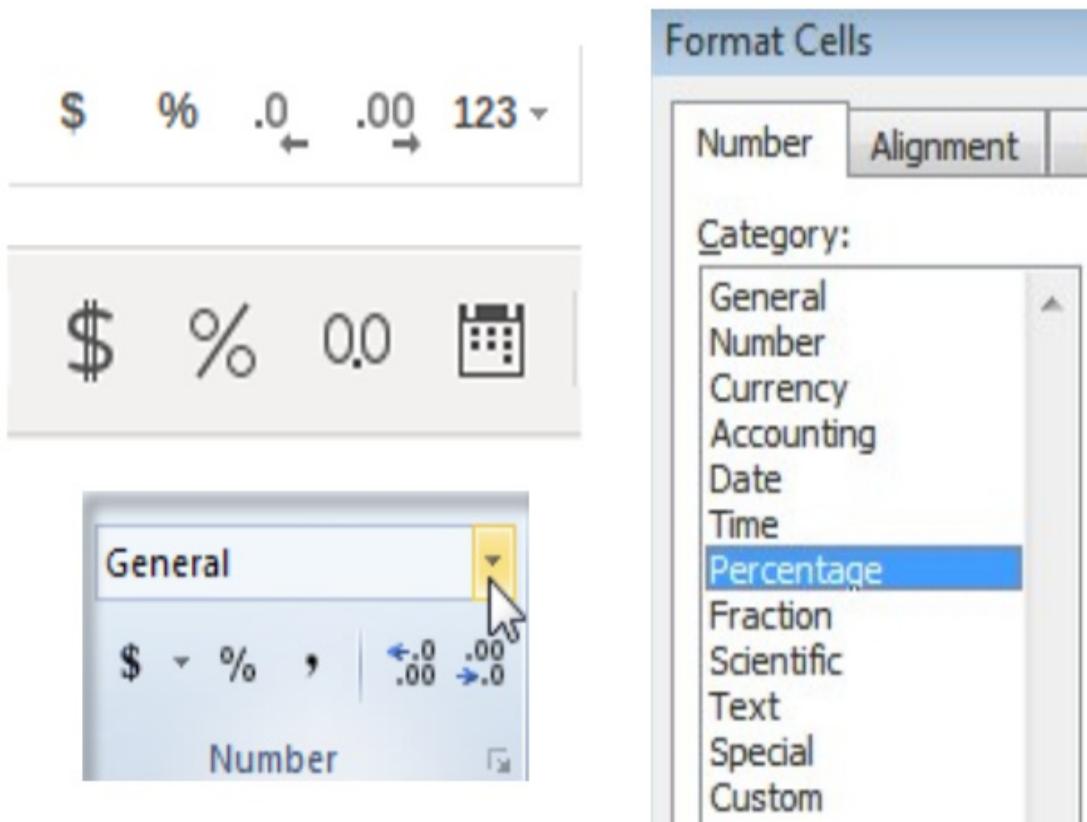
When telling R that a number is an integer rather than a numeric type, there is an alternative to the `L`-suffix syntax. You can explicitly tell R that a number is an integer:

```
myInt <- as.integer(x = 7)
str(object = myInt) ①
#> int 7
```

(1)

**Also an integer**

This has some similarities to the conversion mechanism of spreadsheets, which provide a menu of data types for conversion, some examples of which are shown in figure 2.10.



**Figure 2.10** Various ways to format numbers in a spreadsheet program

There are limits to what R will convert. What if you provide something that clearly *isn't* an integer? You can test this with a few values and see if you get the result you expect:

```
as.integer(x = 3.14)
#> [1] 3
```

Makes sense, perhaps. That's certainly the nearest integer (whole number) to that value. Let's try another:

```
as.integer(x = 2.99)
#> [1] 2
```

Is that what you expected? Depends on how you guessed the function might work. `as.integer()` coerces to the closest integer by rounding toward zero.

What about negative numbers?

```
as.integer(x = -22.6)
#> [1] -22
```

Again the rounding is toward zero.

**CAUTION** This may or may not be what you want to do, so make sure you keep this rule in mind when using this function.

R will do its best to coerce from any type to any other type you ask, but only certain combinations are possible. Integer to numeric is straightforward. Numeric to integer, as you just saw, works, provided you know that it will round toward zero. What about some other types? What about requesting the integer version of a character string involving a negative sign and some unnecessary whitespace?

```
as.integer(x = " -47.8 ")
#> [1] -47
```

①

Note the included whitespace.

You might not be too surprised that a clever enough system could figure this out. As soon as there is something unclear about the string, however, the conversion fails:

```
as.integer(x = "--92.4")
# Warning message:
# NAs introduced by coercion
#> [1] NA
```

This would usually be understood as applying the *unary* (acting on one variable) function of “take the negative value” twice, if it

weren't a string:

```
as.integer(x = --92.4)
#> [1] 92
```

`NA` (for *not available*) is the value produced whenever a conversion isn't possible or a value is missing, and R will usually give you a warning to let you know it's done that.

`as.character()` can convert just about anything because all it needs to do is throw some quotes around the input:

```
as.character(x = 2342.983)
#> [1] "2342.983"
```

## Factor labels

Be careful if you're trying to extract factor levels using these functions. Say you have these factor levels:

```
dozens <- factor(x = c(36L, 24L, 12L, 60L))
dozens
#> [1] 36 24 12 60
#> Levels: 12 24 36 60
```

Now you want to go back to just `integer` types. Though it's reasonable to expect that you could just use `as.integer()`, something goes horribly wrong:

```
as.integer(x = dozens)
#> [1] 3 2 1 4
```

R has given the factor *indices*, not the actual levels. Instead, you need to *first* convert the factor to character to obtain the levels, and then you can convert to `integer`:

```
as.integer(x = as.character(x = dozens))
#> [1] 36 24 12 60
```

Note also that once the values are in `factor` form, they show no distinction between originally being `numeric` or `integer`,<sup>12</sup> so there's no way to explicitly "convert back" to

their original type.

---

<sup>12</sup> The levels are stored as characters, and as you've seen, numeric and integer types look the same when converted to characters.

---

R considers certain values to be convertible to TRUE or FALSE via `as.logical()`. Non-zero numbers convert to TRUE, whereas 0 converts to FALSE. This perhaps makes sense in the context of a signal being either on or off:

```
as.logical(x = 0)
#> [1] FALSE
```

```
as.logical(x = 1)
#> [1] TRUE
```

Most characters/strings convert to NA, the missing value:

```
as.logical(x = "value")
#> [1] NA
```

There are some notable exceptions; the strings T, TRUE, True, and true, as well as F, FALSE, False, and false, each convert to their respective logical values. This can make for some odd results:

```
as.logical(x = "R")
#> [1] NA
```

```
as.logical(x = "S")
#> [1] NA
```

```
as.logical(x = "T")
#> [1] TRUE
```

You can see the difference between `Date` and `POSIXct` by explicitly converting the current system time (which naturally returns an object of type `POSIXct` anyway) into each of these types:

```
as.Date(x = Sys.time())
#> [1] "2018-01-23"
```

```
as.POSIXct(x = Sys.time()) ①
#> [1] "2018-01-23 21:38:47 ACDT"
```

(1)

**The time zone is included in the output. The zone shown here (Australian Central Daylight Time, ACDT) may very well be different from the output in your time zone.**

**CAUTION** In the last example, as `.POSIXct()` (or just `Sys.time()`) showed the date, time (including seconds), and time zone (Australian Central Daylight Time, ACDT, in my case) for the current computer date-time. This will be different whenever/wherever you repeat it. If your computer doesn't have the date, time, or time zone set correctly, this may not accurately reflect these quantities.

## 2.4 Telling R to ignore something

One of the most important components of any piece of programming, be it a full program or just a small snippet of code, is *documentation*. Most programming languages allow for *inline documentation* in the form of *comments*, which are ignored by the system (recall the way that R “reads” your commands, shown back in chapter 1, figure 1.8), but which are very useful to anyone reading the code.

R will ignore anything appearing after a # (including the #) on a line, so you can use this space to document *why* you've made the choice to write your code the way you have. Don't waste your time merely explaining *what* the code does — that should either be obvious (from reading the code itself) or justified with a *why* if it's not.

Exactly where you put comments is a matter of style, but you should aim to keep them as close to the relevant piece of code as possible. You don't need to have a comment for every line, but dividing code into meaningful blocks can help the person reading it (even if that's you).

You may want to start a new line with a #, which means the entire line will be ignored by R and will contain comments. If your comments are too long for the line and *wrap around* to a new line, make sure this too starts with a #. RStudio will often help by filling this in for you.

Comments can also come partway through a line, in which case R will ignore everything from the # onward. This can be handy when you want to make a quick note on a single command, but too long of a line can become difficult to read. When these lines wrap around, the continued comment needs to start with a # also. R has no way to mark a large block of comments across several lines, so each line will need to include its own #. RStudio can help with this though: if you highlight several lines and then press Ctrl-Shift-C, the selected lines will all be commented by inserting a # at the start of those lines (even if it's just one).

Because comment code is ignored, this feature can also be used to “turn off” parts of your analysis code. It can be tempting to rely on this too much, resulting in a large amount of disabled code spread throughout your scripts. It’s much better to delete any code you no longer consider to be active. Making good use of version control (discussed in chapter 1) allows you to *roll back* to older versions of scripts if you want to bring back in features you deleted. Try the examples shown next.

### [Listing 2.2](#) Try it yourself: documenting using comments

```
# Here we are using a=2 because it provides the
# best agreement with the data.      ①
a <- 2

# cosmological constant:
# ratio of dark energy pressure to energy per unit volume
w <- -1

y <- w + a # TODO: cross-check this with formula in
Einstein (1917)      ②

z <- log(x = y) # values of y are assumed to be positive
```

and

```
# non-zero at this point, so this should be  
# safe. (3)  
  
# x <- z - 1 (4)
```

(1)

Note that each comment starts with a #.

(2)

Comments following code should rarely flow over more than one line, but can be made to look neat with careful use of whitespace.

(3)

Comments can serve as useful notes to yourself. Be sure to remove them if they aren't to be shared.

(4)

Inactive (commented out) code should not appear in your final analysis script, but can be a useful way to temporarily remove some code. This line will not be processed (it will be ignored by R).

## 2.5 Try it yourself

Choose a type from those discussed in this chapter, and then enter a value into the Console. Press Enter, and observe the result that appears. Repeat this for more types, and take note of any differences you can find.

Wrap some values in `str()`, and see how R describes them. Wrap more in `typeof()`, and observe the different types. Can you create an integer and assign it to a variable called `my_int`? Can you change it to numeric type and assign it? Should you use a new name? Check the type you've created with `typeof()`.

How many R-acceptable ways can you write 6 billion?

See if you can create a string that R *won't* accept. Can you

figure out why it isn't allowed? Can you fix it?

Can you assign the current date and time to a variable called now? Can you make it only show the month abbreviation and day?

Play with as many of the data types as you can, and see if you can make anything surprising happen.

## Terminology

- *Real number* — A number with a whole and a fractional part (for example, 3.14). R sometimes refers to these as *double* or *numeric*.
- *Integer* — A number with only a whole part, specified by an L suffix (such as 5L).
- *Character* — A text element (for example, "a") containing zero or more symbols. R uses this name as the type of text objects.
- *String* — A series of zero or more characters, possibly including spaces (for example, "I'm learning R!").
- *Logical* — A “truthy” value, such as TRUE or FALSE (or NA).
- *Special character* — A character that's interpreted in a special way inside strings.
- *Escape character* — For example, a slash (\) used to create a special character, specifying that it should be interpreted differently.
- *Factor* — A labelled category level, used when the thing it labels can only take on a certain set of values.
- *Coerce* — To change the type of a variable, potentially automatically.

## Summary

- The computer stores all of its data in binary.

- R has different types of data that it encodes differently.
- R is weakly typed, so you can change variable types if you need to.
- Integers are treated as a distinct type of number, and can be entered with an L suffix.
- Carefully naming variables makes them easier to work with and recall.
- Both types of quotes (single and double) work with strings.
- You can escape characters by prepending them with a slash (\).
- Factors are stored as integers, but have character levels.
- R is case sensitive. so "Abc" and "aBc" are distinct.
- Dates and times can be stored as Date or POSIXct classes.
- Missing values are denoted as NA, for not available.
- Assignment is performed using the <- operator.
- Variable names can't start with a number.
- It's best to explicitly write numbers as integers if that's what you're intending (the L suffix).
- Use double quotes ("x") for strings by default, unless you really need a quote inside a quote (in which case, use single quotes ' "like this "' ).
- Backslashes (\) are used to give special meaning to certain characters in strings.
- Factors are useful but sometimes show up unexpectedly. Use as.character() to extract the labels, not the indices.
- Prevent R from executing code with the comment character (#). Use this to write comments so you remember why you wrote the code the way you did. Comments can appear at any point on a line.

# 3

## *Making new data values*

---

### This chapter covers

- Performing operations between two or more data values
- Comparing values of the same or different type
- How R changes the data type as it needs to

You have your data values, but there's a good chance you'll want to do something with them, like add or multiply. It's time to go back to basics and see how R deals with combining data.

Thankfully, R is a readable language for things like this, and with any luck you'll be able to code up what you're trying to do with the operators you expect. Follow along with this chapter in the Console and try some values yourself to see if you get the results you expect.

### **3.1 Basic mathematics**

The simplest thing you might want to do to two values is add them. No surprises here, the + symbol (*operator*) between two values will do just that, the same as you would on a calculator:

```
2 + 2  
#> [1] 4
```

The same goes for subtraction:

```
4 - 3
#> [1] 1
```

Multiplication in programming tends to use the *asterisk* (\*) rather than a multiply sign (technically,  $\times$ , but commonly seen as an  $\times$ ):

```
3 * 6
#> [1] 18
```

Division uses a slash (/) like you might use in writing a fraction:

```
12 / 4
#> [1] 3
```

One that might not be so obvious is the *exponentiation* operator, which raises a number to a power. This one is used, for example, when you need to square a value. There are two options, though in reality they're different ways of writing the same thing. Here's the first:

```
2 ^ 10
#> [1] 1024
```

The second is much less common and may be confused with multiplication on too fast a skim-read,

```
2 ** 10
#> [1] 1024
```

Notice that no parentheses (()) were needed to group together the digits 1 and 0 into a 10 — R does some guesswork behind the scenes to interpret what you mean, and treats this as the value 10 rather than an invalid expression. The spaces between values and operators are also cleverly interpreted; you could remove them and write 3+2 or 3^2, but for the sake of clarity, it's often best to space things out a little. The less-common variant \*\* does, however, require that there be *no* spaces between the two asterisks.

Dates are a special type in R (recall section 2.1.4), and you can now see why. Although you can certainly subtract dates to calculate a period of time, which R interprets sensibly for

suitably encoded values

```
as.POSIXct(x = "2016-12-31") - as.POSIXct(x = "2016-01-01")
#> Time difference of 365 days
```

adding dates doesn't make sense:

```
as.POSIXct(x = "2016-12-31") + as.POSIXct(x = "2016-01-01")
#> Error: binary '+' is not defined for "POSIXt" objects
```

Somewhat related, R won't let you perform operations between incompatible types for a given operator. You might understand the intention of the following perfectly well:

```
2 + "2"
#> Error: non-numeric argument to binary operator
```

But R won't take the leap of faith in assuming that you mean to add these as numbers. A spreadsheet may have the same apprehensions about adding a number to a string, an example of which is shown in figure 3.1.

	A
1	2
2	"2"
3	=A1+A2

	A
1	2
2	"2"
3	#VALUE!

**Figure 3.1** Attempting to add a number to a string in a spreadsheet, resulting in an error

Spreadsheets may, however, be more flexible when attempting this within a formula, as in figure 3. 2.

	A
1	=2 + "2"

	A
1	4

**Figure 3.2** Attempting to add a number to a string within a formula in a spreadsheet, which doesn't result in an error

R is less fussy over the difference between numeric and integer types, and will allow you to work with these interchangeably if you ask. It returns the result in the most general type possible (in

this case, numeric, which is more general than integer):

```
numPlusInt <- 7 + 3L
str(object = numPlusInt)
#> num 10
```

It should come as no surprise then that certain combinations are out of the question:

```
"7" - "4"
#> Error: non-numeric argument to binary operator
"7" + "4"
#> Error: non-numeric argument to binary operator
```

Others may surprisingly work just fine:

```
as.POSIXct(x = "2016-12-31") + 1 ①
#> [1] "2016-12-31 00:00:01 ACDT"
```

①

**Remember that the time zone shown here (Australian Central Daylight Time, ACDT) is likely to be different from your own output in your time zone.**

Here R has treated the `1` as some number of *seconds* to add to the date-time object on the left side of the `+` operator. To add a whole day, then you need to add 24 hours, each of which has 60 minutes, each of which has 60 seconds:

```
as.POSIXct(x = "2016-12-31") + 60*60*24 ①
#> [1] "2017-01-01 ACDT"
```

①

**Although you certainly mean to be multiplying integers, which you would specify with an L suffix, you can be lazy and multiply numeric values and get the same result.**

## Math with NA

Involving the missing data value `NA` typically leads to more missing data. Trying to add missing data to known data results in the total value going missing:

```
7 + NA
#> [1] NA
NA + 0
#> [1] NA
```

This can lead to some unwanted effects if you have a lot of data values and just one missing value, perhaps because it was converted from another type incorrectly. The `sum()` function, which calculates the sum of its inputs, has an option specifically for this scenario (as do several functions). If you try to take the sum of some values including an `NA` value

```
sum(3, 7, 0, 9, NA)
#> [1] NA
```

the result is missing. But if you tell this function to first remove any `NA` values with the option `na.rm = TRUE`

```
sum(3, 7, 0, 9, NA, na.rm = TRUE)
#> [1] 19
```

the `NA` value is removed before the sum is calculated.

It's best to be suspicious any time you may have `NA` values in your data.

## 3.2 Operator precedence

When multiple operations are acting together, R has rules that determine the order in which they will be evaluated, called *precedence*.

**DEFINITION** *Precedence* refers to the order or ranking of a group. In R, this refers to which operations will be performed before others.

For example, perhaps you've seen the "challenges" floating around social media, with "99% of people won't get this right" posted alongside a short and semi-ambiguous mathematical statement, such as

$$2 + 3 * 0 - 1 = ?$$

These should be no hassle for anyone who remembers their order of operations from school: the acronym PEMDAS stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction — the order in which these operations should be carried out. In the preceding case, given the lack of parentheses (()) or exponents (raised to a power), the next step is to perform the multiplication of 3 and 0, which is 0. This leaves the addition and subtraction steps, which don't depend on their order:

$$2 + 0 - 1 = 1$$

Nonetheless, the comments on said "challenge" will contain any number of answers and people fighting tooth and nail to defend their incorrect justifications.

R has a similar hierarchy for order of operations, but it includes some other operators we haven't discussed yet. Nonetheless, it's good practice to include parentheses where there is doubt about what the intended order of operations is, or to force a particular grouping. In the preceding case, writing this as

$$2 + (3 * 0) - 1$$

would help make it clearer that the 3 is multiplied by the 0 and the result of that takes part in the subsequent addition and subtraction. This becomes essential when you want to

exponentiate to the result of some combination of multiple values:

```
2 ^ (5 * 2)
```

The parentheses dictate that this group needs to be evaluated first, before the exponentiation. This can significantly alter the way an expression is evaluated. Consider for a moment the following odd-looking expression:

```
x <- y <- 3 + 1
```

This is perfectly valid, and R will process it as it understands it (assign the result of  $3 + 1$  to  $y$ , and assign that result to  $x$ ), producing the following:

```
x  
#> [1] 4
```

```
y  
#> [1] 4
```

Including some more parentheses, though,

```
x <- (y <- 3) + 1
```

you can change what the expression means (now assign 3 to  $y$ , then add 1, and then assign that result to  $x$ ) producing

```
x  
#> [1] 4
```

```
y  
#> [1] 3
```

The computer and R have no problems understanding what you've written (a certain way). Do your best to make sure that what you've written is what you mean.

### **3.3 String concatenation (joining)**

You saw earlier that adding two strings ("7" + "4") produced an error, because the + operator works for numbers (numeric or

integer). We could envisage trying to “add” two words together, though, perhaps “butter” and “fly”. But we don’t really mean *add* — we mean *join*, or more strictly, *concatenate*.

For these circumstances, there is a specific function to perform this operation, the `paste()` function, but it has a default of joining strings with a space between them (which is often what you want):

```
paste("butter", "fly")
#> [1] "butter fly"
```

This is the result of the default argument `sep = " "`, which specifies the separator to place between the inputs, with a default of using a space. You can change this to `sep = ""` to remove it entirely, or use the convenience function

```
paste0("butter", "fly")
#> [1] "butterfly"
```

which performs the same operation but with 0 spaces between inputs.

The `paste()` (and `paste0()`) function converts its input to character before pasting together, so you can use other types here too:

```
paste0("value", 31)
#> [1] "value31"
```

You can even use variables you’ve defined, or other `paste()` calls:

```
address_number <- 221
address_suffix <- "B"
address_street <- "Baker Street"

paste(
  paste0(
    address_number,      ①
    address_suffix      ①
  ),
  address_street      ②
)
#> [1] "221B Baker Street"
```

(1)

**Inputs to the innermost paste0()**

(2)

**Input to the outermost paste()**

## Joining NA values

This is one scenario in which a missing value can become non-missing, perhaps unexpectedly. By default, the inputs to `paste()` are converted to type `character` with the `as.character()` function. But that function preserves missing values, so these two are different results:

```
as.character(x = NA)      ①
#> [1] NA    as.character(x = "NA")      ②
#> [1] "NA"
```

(1)

**A missing value**

(2)

**The string "NA"**

You might expect then that `paste()` would produce a missing value when one of its inputs is `NA`, but it handles this smoothly:

```
paste("a missing value is denoted", NA)
#> [1] "a missing value is denoted NA"
```

This is noteworthy enough to receive special mention in the `help()` page for `paste()`: “Note that `paste()` coerces `NA_character_`, the character missing value, to ‘NA’ which may seem undesirable, e.g., when pasting two character vectors, or very desirable, e.g. in `paste('the value of p is`

' , p )."

Enter some more values into the Console with combinations of these operators and make sure you’re comfortable with them. In the next section, you’ll see how to make comparisons between values.

### 3.4 Comparisons

The essence of scientific results boils down to comparing values. Are there fewer  $y$  than a decade ago? Has  $z$  grown this week? Is  $j$  faster than  $k$ ? Are any of  $m$  significant effects (fitting some criteria)? If we didn’t require comparisons between data, coding analyses would be fairly straightforward — always do this, then that, then the other thing. Because we do require comparisons, though, we must know how to tell R to compare values.

The result of a comparison will always be a *logical* value (recall from section 2.1.5): either TRUE or FALSE (or missing: NA). The simplest comparisons you can make are “Is  $x$  greater than  $y$ ?” ( $x > y$ ) and “Is  $x$  less than  $y$ ?” ( $x < y$ ), using the “greater than” and “less than” operators,  $<$  and  $>$ :

```
7 > 4
#> [1] TRUE
```

```
9 < 3
#> [1] FALSE
```

**CAUTION** Recall the example in section 2.2.3, which accidentally included a space in  $<-$  and produced a comparison. Be very careful with spaces. Whitespace can (and should) be included around the comparison operators to help make the code clear. It’s all too easy to miss the difference between  $x < -3$  and  $x <- 3$ .

You can also allow for the possibility of “Is  $x$  greater than or

equal to  $y$ ?” ( $x \geq y$ ) and its partner “Is  $x$  less than or equal to  $y$ ?” ( $x \leq y$ ):

```
5 >= 6
#> [1] FALSE
```

```
3 <= 3
#> [1] TRUE
```

You can test whether two numbers are equal to each other with a “double equals” ( $==$ ), which asks “Is  $x$  the same value as  $y$ ?” ( $x == y$ ). The opposite question can be asked also: “Is  $x$  a different value than  $y$ ?” or “Does  $x$  not equal  $y$ ?” ( $x != y$ ):

```
3 == 3
#> [1] TRUE
```

```
7 != 4
#> [1] TRUE
```

Some care needs to be taken with these operators. They test whether two values appear to be the same, but not whether they are *precisely* the same. Even though you can specify an integer-like value as either an integer or a real number, these are stored differently. They represent the same value, though, so  $==$  treats these as equal when testing:

```
5L == 5
#> [1] TRUE
```

If you truly want to test whether these are the same thing, the `identical()` function checks the types of the input values before declaring things identical:

```
identical(x = 5L, y = 5)
#> [1] FALSE
```

## Comparisons between real values

Be very careful when comparing non-integer (real) numbers against each other. Recall from chapter 2 that

computers store data in binary (as ones and zeroes). Because of this, there is a limitation to the precision with which decimal numbers can be stored. You may enter the value  $0.3$ , but the computer attempts to store as many digits (zeroes in this case) at the end of that as it can. In the same way that you can't really write  $1/3 = 0.3333\dots$  exactly without going on forever, computers can't store values that aren't exact powers of 2 without being off by a little.

You can see this in effect by requesting more digits in the output from a `print()` command:

```
print(x = 0.3, digits = 17)
#> [1] 0.29999999999999999
```

When mathematical operations take place, these extra or lacking bits also contribute. So although you may see a nice rounded value in your output, in terms of the digits the computer knows about, it may be slightly off from that, leading to unexpected results:

```
print(x = 0.1 + 0.2)
#> [1] 0.3
print(x = 0.1 + 0.2, digits = 17)
#> [1] 0.30000000000000004
```

This isn't unique to R.<sup>1</sup> Rather, it's inherent in any computer language. The safest way to avoid this issue is to never compare real numbers against each other if you need the answer to be exact:

---

<sup>1</sup> See the article "Floating Point Math" at <http://0.30000000000000004.com>.

```
0.1 + 0.2 > 0.3
#> [1] TRUE
```

The `!` in `!=` represents the notion of *not*, as in *not equal*. This can appear in several different places with the same effect:

```
3 != 4
#> [1] TRUE
```

```
!(3 == 4)
#> [1] TRUE
```

```
(! 3 == 4)
#> [1] TRUE
```

What if you try to compare to a missing value? The comparison of anything to the missing value `NA` is `NA`

```
3 > NA
#> [1] NA
```

```
7 == NA
#> [1] NA
```

even if the comparison involves a “not” operator:

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

Comparison between the two logical values can be summarized in a *truth table*, where the intersection of a row and a column shows the result of the comparison `row == column`, as shown in [table 3.1](#). Notice that comparing anything to `NA` yields `NA`, including `NA` itself.

**Table 3.1** Truth table for the equals operator (`==`)

<code>==</code>	<code>TRUE</code>	<code>FALSE</code>	<code>NA</code>
<code>TRUE</code>	<code>TRUE</code>	<code>FALSE</code>	<code>NA</code>
<code>FALSE</code>	<code>FALSE</code>	<code>TRUE</code>	<code>NA</code>
<code>NA</code>	<code>NA</code>	<code>NA</code>	<code>NA</code>

Comparing anything to `NULL` results in nothing, but a particular “type” of nothing — still a logical value, but one that is of length 0:

```
3 > NULL
#> logical(0)
```

```
TRUE == NULL
#> logical(0)
```

```
NA != NULL
#> logical(0)
```

There are other ways to compare logical values; you can also combine them with *and* (`&`) and *or* (`|`) operators. Let's make another truth table for the logical options, as shown in [table 3.2](#) for *and*.

**Table 3.2** Truth table for the *and* operator (`&`)

&	TRUE	FALSE	NA
TRUE	TRUE	FALSE	NA
FALSE	FALSE	FALSE	FALSE
NA	NA	FALSE	NA

The surprising result of [table 3.2](#) is the combination of `NA` and `FALSE`, which yields `FALSE`. The help page for `?`&`` explains: “`NA` is a valid logical object. Where a component of `x` or `y` is `NA`, the result will be `NA` if the outcome is ambiguous.”

A similar table can be constructed for the *or* (`|`) operator, as shown in [table 3.3](#). Again, comparison with `NA` doesn't necessarily lead to `NA` as long as the comparison is unambiguous.

**Table 3.3** Truth table for the *or* operator (`|`)

	TRUE	FALSE	NA
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NA
NA	TRUE	NA	NA

## Individual comparisons

Although it may be tempting to think that you can write

logical combinations the way you might read them, it's important to remember that only one comparison can be made at a time. To test whether some variable  $x$  was greater than 3, you would write the following:

```
x <- 4
x > 3
#> [1] TRUE
```

To also test whether  $x$  was less than 5, you might try adding in this condition as "also less than 5":

```
x > 3 & < 5
#> Error: unexpected '<' in "x > 3 & <"
```

This fails because when R reads in this expression, it breaks it down into the appropriate levels of precedence (recall section 3.2), and  $<$  has a higher precedence (is evaluated earlier) than  $\&$ ; but here  $<$  has nothing to compare to, so the call fails.

Instead, you need to repeat a little and make the two comparisons explicit:

```
x > 3 & x < 5      ①
#> [1] TRUE
```

①

**Equivalent to  $(x > 3) \& (x < 5)$**

There are also double forms of *and* ( $\&\&$ ) and *not* ( $\|$ ) operators that evaluate, left to right, one element at a time, until a result is definitely reached. This becomes important and useful when you have collections of logical comparisons to test, but only require a single answer. You may need to make a thousand  $\&$  comparisons to test whether all the values of two inputs are the same; but if they differ in the fourth pair then they're *definitely* not the same, so the rest of the comparisons can be skipped.

## Testing whether a value is equal to NA

If you need to compare whether a value is `NA`, the handy built-in function `is.na()` returns `TRUE` if the input is `NA`, and `FALSE` otherwise:

```
is.na(x = 7)
#> [1] FALSE
is.na(x = NA)
#> [1] TRUE
```

## 3.5 Automatic conversion (coercion)

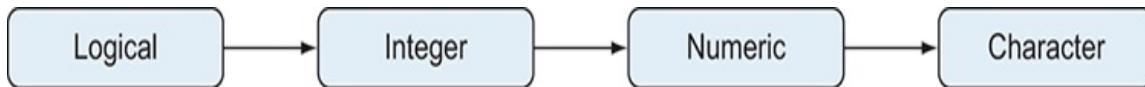
Occasionally, R will perform a conversion on your behalf (whether you wanted it to or not). Typically, this is a useful feature; to add an integer and a non-integer, you might try this:

```
3L + 2.5
#> [1] 5.5
```

The result can be seen by examining the structure:

```
str(object = 3L + 2.5)
#> num 5.5
```

What has happened here? The C code underlying the `+` function has a few lines specifically for this scenario. If *one* (and only one) of the arguments (with the shorthand `+` notation, the left or right side) is of type `numeric` and the other is of type `integer`, the integer value is *coerced* to a numeric value, and then the two are added, producing a final numeric value. This is the result you want — converting an integer to a numeric value is essentially trivial: no rounding is required. Converting from a numeric value to an integer, however, has some potential complications, as you saw in section 2.3. If automatic coercion takes place, R will always coerce to the more general structure. The ordering from most specific to most general is shown in figure 3.3.



**Figure 3.3 Ordering of coercion**

R considers certain values to be equivalent to TRUE and others equivalent to FALSE. For numbers, the value 0 (or 0L) converts to FALSE, whereas any other (positive or negative) number converts to TRUE. You can observe this by requesting the conversion explicitly:

```
as.logical(x = 0)
#> [1] FALSE
```

```
as.logical(x = 1)
#> [1] TRUE
```

Or compare with the approximate test ==:

```
0 == FALSE
#> [1] TRUE
```

```
1 == TRUE
#> [1] TRUE
```

The automatic conversion the other way means you can add logical and real values together:

```
8 + TRUE
#> [1] 9
```

This becomes very handy for counting binary values:

```
sum(TRUE, FALSE, TRUE, TRUE)
#> [1] 3
```

Performing comparisons is when you are most likely to encounter automatic coercion. You may not expect the following to work at all, but it does:

```
"a" > 5
#> [1] TRUE
```

When R notices you're trying to compare two different types, it performs coercion to the most general type — in this case,

character is more general (it's the most general) than numeric. What happens next depends on where you live; the strings/characters are compared by their place in the encoding scheme's table. This is language-dependent (which tends to depend on the *locale* your computer recognizes): this example is using (Australian) English with a UTF-8 scheme (`en_AU.UTF-8`), for which letters come *after* numbers. The `help()` page for the comparison operators (for example, `?`>``) notes that in Estonian, *Z* comes between *S* and *T*,<sup>2</sup> so be aware when using this feature that your results may differ from someone else's.

---

<sup>2</sup> Read more about this in the Wikipedia article at <http://mng.bz/5Yx8>.

---

This is also the reason the wayward space in the assignment operation back in section 2.3 produced a result. Recall that you tried to assign the value 3 to the variable `a` but accidentally inserted a space inside `<-`, and so instead generated a logical value since `a` already held the value "x":

```
a <- "x"
a <- 3
#> [1] FALSE
```

Here R sees a comparison between two different types, so R converts to the most general type (in this case, character) and checks the encoding scheme's table to see if "x" comes before or after -3. In my locale, letters come *after* numbers, so this comparison returns FALSE.

You could reasonably expect that the following is automatically coercing to numeric

```
"2" < "3"
#> [1] TRUE
```

which would produce the output as shown. A simple change shows that this isn't the case:

```
"2" < "13"
#> [1] FALSE
```

In these cases, no coercion is required, because both values are already of type `character`. They are again compared via their positions in the encoding table, and these are sorted numerically such that "13" (starting with a "1") is before "2".

## 3.6 Try it yourself

Daily temperatures are typically found in one of two scales: Celsius and Fahrenheit. It's useful to know how to convert between the two. If you have a temperature in Fahrenheit and want to convert it to Celsius, you can enter the Fahrenheit value into this expression to find the Celsius value:

```
Celsius <- (5 * Fahrenheit - 32) / 9
```

So, if you have

```
temp_F <- 88
```

you can calculate the temperature in Celsius using

```
temp_C <- 5L * (temp_F - 32L) / 9L
```

(the `L` specifications aren't essential, but are good practice). You can now examine what value this takes:

```
temp_C  
#> [1] 31.11111
```

Going the other way, you can invert the expression:

```
Fahrenheit = (9 * Celsius / 5) + 32
```

So you should be able to obtain the original value again:

```
temp_F_recalculated <- (9L * temp_C / 5L) + 32L  
temp_F_recalculated  
#> [1] 88
```

You can reassure yourself that this is in fact the same value:

```
temp_F == temp_F_recalculated  
#> [1] TRUE
```

Evaluate the preceding conversions (or better yet, write your own) and convert  $0^{\circ}\text{C}$  to Fahrenheit. Convert  $100^{\circ}\text{F}$  to Celsius. Convert  $-40^{\circ}$  from one scale to the other.

## ***Terminology***

- *Operator* — A symbol representing an operation to be performed on one or more values: for example, `+`
- *Precedence* — The order in which operations will be evaluated
- *Expression* — A command for R to interpret; performing an operation on 0 or more data values

## ***Summary***

- You can use R as a calculator more or less as you'd expect.
- Certain types of data can't be added/subtracted.
- Operators have different precedence, which can be overridden with parentheses.
- Strings can be combined using `paste()`.
- Coercion will result in data being converted to the most general type.
- `character` is the most general type of data.
- Text can be compared to numbers because the latter will be coerced to `character`.
- Comparisons between real numbers is dangerous due to rounding differences.
- Comparing `NA` to anything produces `NA`, even `NA` itself.
- Strings can be compared, but doing so depends on your computer's settings, usually dependent on where you live.

# 4

## *Understanding the tools you'll use: Functions*

---

### This chapter covers

- Repeating operations with functions
- The boundaries of where things are defined (scope)
- Messages, warnings, and errors, and how to deal with them
- Testing that your code does what you think it does

The true power of a programming language arises when you can tell the computer to do something and that request is simpler than just doing the work ourselves. Any time you want to calculate something more than once, it's potentially a good idea to wrap the code up into a meaningful expression that can be reevaluated with a different (or even the same) input.

You'll quickly see that *functions* are the standard method of achieving this goal in R. Once errors inevitably arise, you'll also see how to manage things that don't go according to plan.

### **4.1 Functions**

A *function* (in programming) refers to a series of defined operations or routines that achieves some task, usually involving data, that produces some result. Functions can have input data

(or not) and produce output data (or not) or perform some action. Functions may also define or use more data if the data is defined by the function itself (for example, constant numbers or values).

In R, functions are handled in the same way that data is — stored with a variable name — and can be sent as input to other functions. Saving the operations in a named function (the *body* of a function) has many advantages, including the following:

- *Abstraction* — The function can be provided with different data.
- *Reusability* — It reduces copy-and-paste repetition of the same operations, which can lead to errors.
- *Resilience* — Checks can be performed on the input data to ensure it meets the assumptions of the operations to follow.

R is a *functional* language;<sup>1</sup> it uses *functions* to interact with data, returning some other output and otherwise leaving the input unchanged (as opposed to a *procedural* language, which uses *statements* to modify data with an output dependent on the current state of the system). In a functional language, data is never really changed, and this holds in R.

<sup>1</sup> An *impure* one, in that not all functions provide *referential transparency* (where an expression could be replaced by its output value) and many functions produce *side effects* (such as plotting).

<sup>2</sup> For the most part. The `data.table` package bypasses this restriction by acting on data in-memory, which greatly increases the speed of its operations.

To repeat an operation — perhaps generalize it to arbitrary inputs — you can use a function. If you frequently need to calculate the area of a circle, you can do that each time with the right *formula* (a similar enough concept to a function). The formula for the area of a circle is

$$A = \pi r^2$$

where  $\pi = 3.14159\dots$  is a built-in constant in R named `pi`. To calculate an area  $A$ , all you need to know is the radius  $r$ , which you square (multiply by itself) and then multiply by `pi`. You could do this several times for radius values of 3, 4, and 5. Recall that you can raise a value to a power with the `^` operator, so to calculate the area ( $A$ ) of circles with these radii, calculate  $\pi$  multiplied by the radius ( $r$ ) to the power of 2:

```
pi * (3 ^ 2)
#> [1] 28.27433
```

```
pi * (4 ^ 2)
#> [1] 50.26548
```

```
pi * (5 ^ 2)
#> [1] 78.53982
```

Or you could create a function that does this for you if you supply the radius. Functions are stored as objects in R, so you name them using the assignment operator like any other object/variable.

Although it has been straightforward enough to enter simple commands into the Console, more complex commands become a bit too unwieldy to enter that way. It's usually better to work with a *script* in the Editor pane.

Create a new script file by clicking File > R Script. Enter the following code into the Editor pane:

```
areaCircle <- function(radius) {
  area <- pi * radius ^ 2
  return(area)
}
```

①

The keyword `function()` tells R that you're defining a function. This is a function of the input `radius`, which will be allowed to vary. The body of the function is bounded by `{` and `}`.

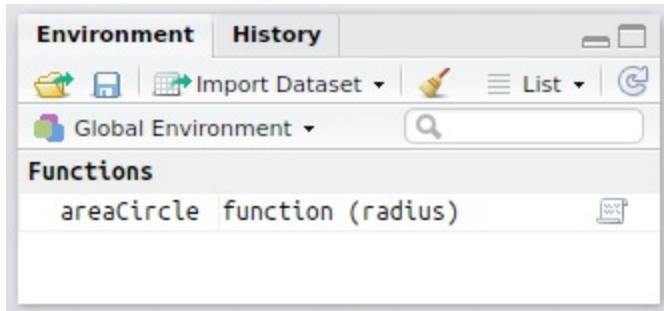
②

You can create (temporary) variables inside functions — you'll see why these are temporary in section 4.1.8.

(3)

A function returns (outputs) a value that is the result of the calculation.

Highlighting the function definition and evaluating it (by pressing Ctrl-Enter or Cmd-Enter) makes the function available for use, and it now appears in the Environment listing as shown in figure 4.1.



**Figure 4.1 Function defined**

You can now calculate the area of a circle using only the minimal information (a radius value) and the function. When you evaluate this function with a value as the `radius` argument value, R performs the operations stored in the *body* of the function, and then *returns* (outputs) the value identified by the `return()` call. By default, outputs are printed, so the results of the calculations are printed to the Console:

```
areaCircle(radius = 3)
#> [1] 28.27433
```

```
areaCircle(radius = 4)
#> [1] 50.26548
```

```
areaCircle(radius = 5)
#> [1] 78.53982
```

In this way, you can *abstract* away calculations and name them appropriately; you don't need to explicitly write out the area

formula each time you want to use it; you now have a shortcut to that in the form of a function. In the preceding example, the function is named according to what it calculates (the area of a circle); you'll likely be calculating many different things, so take some care in how you name your functions.

You have now calculated areas, but where are the values? Notice that nothing more has appeared in the Environment listing. Because of the way R is designed (*functional*, leaving data unchanged), evaluating a function doesn't change *any* data, so no new data is created in any permanent sense. It's printed, and then it's gone.<sup>3</sup> In order to retain the value calculated, you need to assign it to another variable.

---

<sup>3</sup> With the exception that it can still be recalled immediately using `.Last.value`, as discussed in section 2.2.1.

## On assigning the returned value

A common complaint from people having trouble with their code is "my function doesn't do anything." These complaints typically involve function calls on data that look like the following:

```
my_data <- 7
my_data
#> [1] 7
increase_my_data <- function(data_input) {
  updated_data <- data_input + 1
  return(updated_data)
}

increase_my_data(data_input = my_data)
#> [1] 8
my_data # (not changed)
#> [1] 7
```

Can you spot what is missing? `increase_my_data` should return something very different from its input — and it does, but this person hasn't assigned that returned value to

anything. R evaluates `increase_my_data(my_data)` and prints the value to the Console (by default), but doesn't update the variable `my_data`; that would be a *procedural* way of doing things, whereas R works in a *functional* manner, leaving data alone.

The "solution" is usually to assign (recall section 2.2) the output (return value) of `increase_my_data`, in this case back to the original data name:

```
my_data <- increase_my_data(data_input = my_data)
my_data
#> [1] 8
```

Keep in mind that this overwrites the value `my_data`, which may not always be the result you are trying to achieve.

You've seen how a simple function is written and used, but what's actually happening? Building a proper understanding of how R sees functions can greatly improve your ability to use these valuable tools.

### **4.1.1 UNDER THE HOOD**

You may write the expression

```
x <- 2
x <- cos(x = 2*x)
```

and think that the object `x` is being updated from some value (2)<sup>4</sup> to the cosine of twice that value,<sup>4</sup> but there's a lot more going on behind the scenes here. In reality, the value `2*x` is provided to the `cos()` function as an input. The `cos()` function is some series of operations to perform involving this input, resulting in a *return value*. This value is then assigned to the variable `x`, overwriting the previous value. Right up until that last step, though, the value of `x` was completely untouched. You can confirm this by looking at the value before and after calculating `cos(x)`, but not assigning it back to `x`:

---

<sup>4</sup> From trigonometry; together, the sine (`sin()`), cosine (`cos()`), and tangent (`tan()`) functions can be used to describe, for example, the ratios of sides of a triangle

---

```
x <- pi    ①
x
#> [1] 3.141593

cos(x = 2*x) ②
#> [1] 1
```

①

pi is one of the few built-in constants.

②

The cosine of 2\*pi should be 1.

Although `x` appears in this function call, it still has the value you originally assigned to it:

```
x
#> [1] 3.141593
```

An equivalent set of expressions to the `x <- cos(x = 2*x)` example is the following:

```
.y <- cos(x = 2*x)
x <- .y

x
#> [1] 1
```

Here, `.y` becomes inaccessible directly. You can shorten this to just `x <- cos(x = 2*x)` safely because of the way R performs these operations. Recall that due to the *REPL* nature of R, once you press Enter the underlying code takes over and begins digesting your expression. For the previous example, the *functions* are unpacked.

## On R functions

To understand computations in R, two slogans are helpful:  
 Everything that exists is an object. Everything that happens is a function call.

*—John Chambers, creator of the S programming language and core member of the R programming language project*

You therefore have a neat and tidy way of asking R to calculate the cosine of double some value, and then assign this result back to the original variable. But under the hood, R treats these three operations as *functions*:

```
cos(a)      ①
`*`(e1, e2)  ②
`<-`(l, r)   ③
```

①

**Input: angle a. Output: cosine of a.**

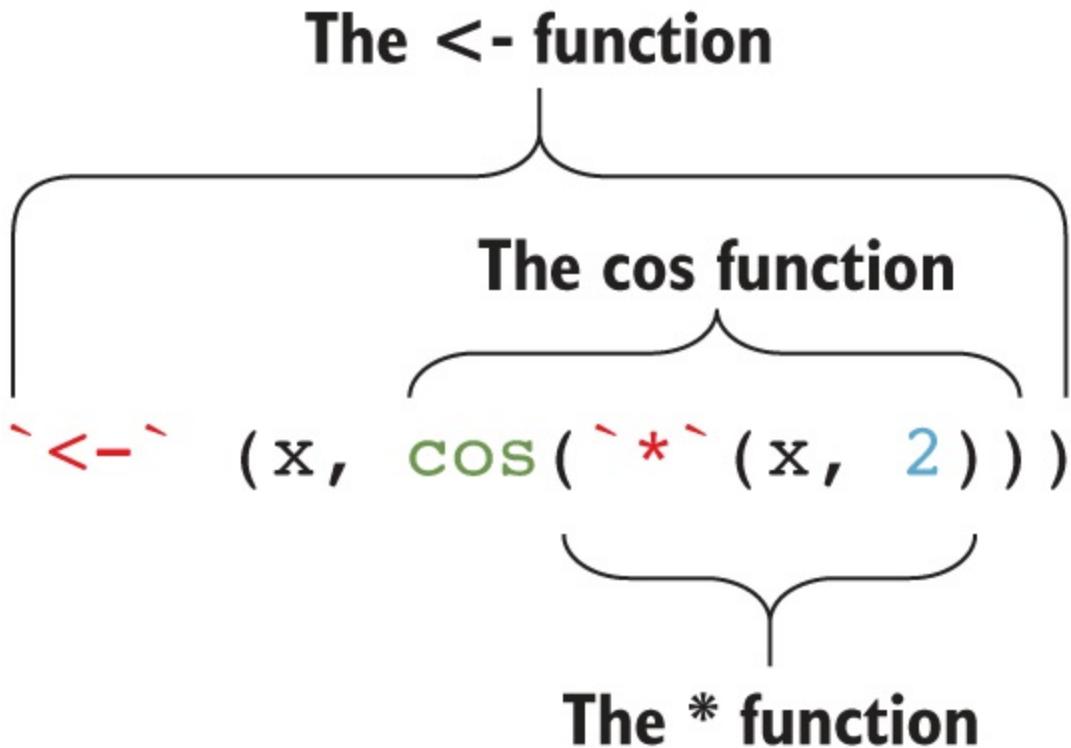
②

**Input: values e1 and e2. Output: product of e1 and e2, that is, e1 \* e2.**

③

**Input: expressions l and r. Output: none, with side effect of assigning the value r to the variable l.**

The tidy little expression is really the *composition* of the three functions, evaluated from the inside outward: the `\*` function with inputs x and 2 produces a value, which becomes the input to the cos() function, which itself produces a value, which becomes the second input to the `<-` function, as shown in figure 4.2.



**Figure 4.2 Function composition**

The backticks (`) tell R you're calling a function directly, so it can skip over trying to unpack this again. The `<-` (assignment) function doesn't return anything (visibly, at least),<sup>5</sup> so nothing is printed to the Console when this expression is evaluated. Up until the outermost function, it is the return values of the inner functions that are evaluated in the next outermost function, and `x` will remain unchanged.

---

<sup>5</sup> Technically, the assignment operator returns a value *invisibly* (not printed to the Console), so it's still there even though you don't see it

You won't be calling functions in this manner, but it's very useful to keep in the back of your mind that this is what R is really seeing.

### 4.1.2 FUNCTION TEMPLATE

The general structure of a function definition in R is the following. The function is assigned to a variable just like a value

would be. In this case, the variable name is `name_of_function`:

```
name_of_function <- function(argument1, argument2) { _____ ①
  temporary_variable1 <- other_function(argument1) _____ ②
  final_variable <-
  yet_another_function(temporary_variable1)
  return(final_variable) _____ ③
}
```

①

**Arguments are inputs to a function. Placing the function body-enclosing brackets on this line or the next is a matter of style. They can even be left off entirely if a function body contains only a single expression.**

②

**Operations are performed on the inputs and potentially other data with more functions and assigned to temporary variables (which functions are possible and why these are temporary variables will be covered later in this chapter).**

③

**The output of the function, the value that will be returned by calling the function, is identified by a special function called `return()`. Alternatively, R will assume that the last expression in a function produces the value to return.**

You *call* a function (possibly with *arguments*) by adding parentheses surrounding any arguments (or no arguments) to the name of the function. (I cover how arguments work and how to define them in section 4.1.3.) So, the previous template can be called using the following:

```
name_of_function(argument1 = x, argument2 = y)
```

Recall from section 2.2.3 that the value(s) given to arguments

is/are specified with `=` rather than `<-`. If you had another function that took no arguments (or they had defaults), then you could omit any arguments, but the parentheses remain:

```
anotherFunction()
```

Functions can be used whenever you might want to repeat code; to group or isolate some steps of your analysis (perhaps to make it clearer what the purpose of those steps is); or to manipulate data when you're only interested in preserving the final result, not the intermediate transformations created along the way.

## Autoprinting return() values

If no `return()` function is explicitly stated at the end of a function, the result of the last expression within the function is used as the return value. If the result of evaluating a function is not assigned to anything (the function is just evaluated: for example, `myFunction(7)`), then typically this will result in the return value being printed in the same way as evaluating a variable would. A special case is when the last expression in a function returns no value (or does so invisibly), such as the assignment operator `<-`. In these cases, evaluating the function without assigning the result will produce no output (but the return value can still be assigned).

This example prints the value to the Console:

```
returningFunction <- function() {  
  2 + 7  
}  
returningFunction()  
#> [1] 9
```

But this example doesn't:

```
nonReturningFunction <- function() {  
  x <- 2 + 7  
}
```

```
nonReturningFunction()
```

These two are equivalent, however, if you assign the results:

```
a <- returningFunction()
b <- nonReturningFunction()
a == b # are these equal?
#> [1] TRUE
```

In either case, using `return()` explicitly is highly recommended, especially as it allows you to `return()` a value from the middle of a function, rather than the end:

```
bigOrSmall <- function(x) {
  if (x > 10) {
    return("x is big")
  } else {
    return("x is small")
  }
}
```

(1)

I'll cover what `if()` does in chapter 8, but perhaps you can already guess.

(2)

**Exits the function from the middle, when x is greater than 10**

(3)

**Exits the function normally, at the end, when x is not greater than 10**

The function returns one of the two possible results, returning normally

```
bigOrSmall(x = 8)
#> [1] "x is small"
```

or at another designated point

```
bigOrSmall(x = 12)
#> [1] "x is big"
```

In R, because “everything that happens is a function call,” even seemingly simple things like the addition operation `+` are functions behind the scenes, just cleverly written. The `+` operator is shorthand for a function that takes two arguments; the code behind R recognizes when this is being used and performs the translation for you. You can see this by evaluating `+` without the parentheses, which returns the function body itself. Type the following into the Console, and press Enter:

```
`+`  
#> function (e1, e2) .Primitive("+")
```

So `2 + 2` is the function call (backtick escaped):

```
`+`(2, 2)  
#> [1] 4
```

## Just a simple function?

Don’t let the apparent brevity of `function (e1, e2) .Primitive("+")` fool you. Behind the scenes are a whopping 116 lines of C code (which call further functions too) that handles all the different possible things you might mean when you place a `+` between two other things. R nicely abstracts all of that away so you only need to write the `+`, whether you’re adding a numeric type to an integer type, or one of the values is the missing value `NA`.

It can even handle the case when you don’t provide *anything* as the first argument, in which case the underlying code ensures that the expression is understood as a *unary* operation (single argument, as opposed to a *binary* operation with two) and gives you back the original value. Nothing interesting there, but the same code handles the arithmetic for the `-` function, in which case the *unary* operation returns the value with the opposite sign.

We haven’t discussed more complicated structures yet, but this same chunk of code handles those too.

The vast majority of R functions are vulnerable to being overwritten by a user in their session, so once again naming things becomes a challenge. There's nothing stopping you from deciding to name a function `+` and deciding that it should do the same thing as the already defined `-` function:

```
`+` <- ` -`
```

This of course wreaks havoc with sensibilities and anything you may want to do involving numbers down the track:

```
2 + 2
#> [1] 0
```

Clearly this was a poor choice of function name. If you do this temporarily, it's a very good idea to return things to their normal working order:

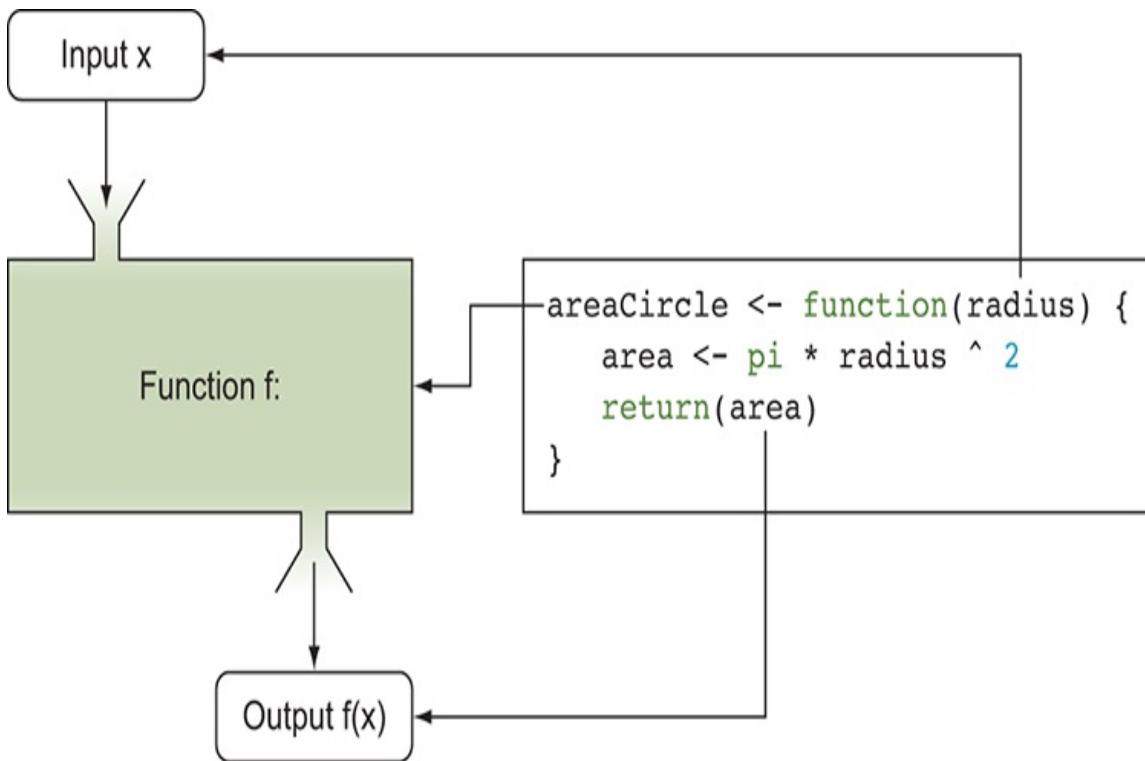
```
`+` <- base::` +`
```

Every operation you perform will be a combination of one or more functions. If you want those functions to know about your data, you need a way to communicate values. You do that via the function *arguments*.

### 4.1.3 ARGUMENTS

Now that you know what a function looks like, how do you give a function some data to work with? Here it may be helpful to think of a function as a mechanical processing machine; it takes some input (the raw materials), performs some operations, and then outputs something, as shown in [figure 4.3](#).

The input could vary, and, depending on the internals of the machine, so might the output. The way R handles this is to create a new variable for each input that is specific to the function. These are just like the other variables we have been working with, except that (as you'll see in section 4.1.8) they exist only in their own little world within the function.



**Figure 4.3** The function machine (adapted from Wvbailey [Public domain], via Wikimedia Commons)

The value of this is *abstraction*: the argument variable now represents some unknown input data that the function needs to work with, and that input data could be different each time the function is evaluated. This concept is present in spreadsheet programs when using formulas; the arguments there are usually named in an abstracted way also, as in figure 4.4.

	A	B	C	
1		4		
2		8		
3	=sum()			
4		SUM(▶ number 1, number 2, ... )		
5				

**Figure 4.4** Arguments to the `sum()` function in a spreadsheet, named `number 1` and `number 2`

Let's make the machine analogy a little more specific and imagine that the machine takes a word, performs some operations on it, and returns the first letter. A function that performs this operation might look like the following — enter

this into a script in the Editor pane of RStudio, and evaluate it:

```
first_letter <- function(word) {
  ## extract the first letter of the word using substring()
  ## which is called as substring(text, first, last)
  ## see ?substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(first)
}
```

Here a function called `first_letter` takes one argument — a word. This is the *input* (argument) to the function. You can evaluate this function with any input value:

```
first_letter(word = "cat")
#> [1] "cat starts with c"
#> [1] "c"
```

The first output (beginning with [1]) is the output from the `print()` function. The next output is the value returned from `first_letter()`. The internally created variables `first` and `message_text` aren't available outside of this function; they only exist while the function is being evaluated. So, the following generates an error:

```
message_text
#> Error: object 'message_text' not found
```

Similarly, you haven't created a variable `word` outside of the function, even though you used an `=` to provide the value to the argument:

```
word
#> Error: object 'word' not found
```

You also haven't assigned the return value to anything, so evaluating the function merely prints the result. You could, however, use the function to create a new variable by assigning the result :

```
first_of_apple <- first_letter(word = "apple")
#> [1] "apple starts with a"
```

Note that the message was printed during evaluation, but the result was not printed. You can retrieve the value by printing it:

```
first_of_apple
#> [1] "a"
```

A function is only useful if you need to use it more than once; otherwise you could just write its contents out and be done. To process several words and save their first letters to new variables, use a little repetition:

```
first_of_cat <- first_letter(word = "cat")
#> [1] "cat starts with c"
```

```
first_of_dog <- first_letter(word = "dog")
#> [1] "dog starts with d"
```

```
first_of_fish <- first_letter(word = "fish")
#> [1] "fish starts with f"
```

This function was short, but hopefully you can appreciate that abstracting longer series of operations has great value. As it stands, you have made the function fairly general — it is able to extract the first letter of a word, and you don't need to write out all the steps to do so each time.

## **4.1.4 MULTIPLE ARGUMENTS**

You aren't limited to just a single argument, though; you can have as many as required. The processing done between input and output may involve several named or unnamed arguments. There are a couple ways to do this: either pass in a more complex object as the input (we'll get to how to do this later), or pass in several arguments. The latter option is safer because it's easier to

track what the inputs are.

Functions aren't limited to inputs of strings; you could have numbers, dates — any type of R object, or even more functions. This is a powerful feature of R that you'll be thankful you learned.

If a function knows what to do with more arguments, it will use them. If it receives an argument that it doesn't know how to process, it will produce an error:

```
first_letter(word = "banana", music = "jazz")
#> Error: unused argument (music = "jazz")
```

This gets somewhat trickier when functions use the “dot-dot-dot” syntax (...), which can process *any* arguments, so just be extra careful in that case.

The opposite is also worth mentioning; just because a function *has* some arguments doesn't mean you absolutely have to use them. This is highly dependent on the function (particularly, whether it really *needs* certain arguments). You could create a trivial function that takes an argument, does absolutely nothing with it, and returns the number 3:

```
trivial_function <- function(arbitrary_argument) {
  return(3)
}
```

Calling this with or without the argument works just fine:

```
trivial_function(arbitrary_argument = 7)
#> [1] 3

trivial_function()
#> [1] 3
```

R will stop evaluating arguments as soon as it can. This is known as *lazy evaluation*, and it both helps speed up code and makes it more flexible. If you add another argument that isn't required and isn't used, say

```
lazy_function <- function(x, y) {
```

```
    return(x)
}
```

then R doesn't even mind if you try to use a variable that doesn't exist in the second argument, because it never attempts to evaluate that variable:

```
lazy_function(x = 2, y = undefined_variable) ①
#> [1] 2
```

①

**undefined\_variable has not been defined, but R won't even try to evaluate it.**

If you expand the function to include some processing of the output, you can pass in a function name as well:

```
first_letter_formatted <- function(word, fun) { ①
  ## extract the first letter of the word using substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(fun(first)) ②
}
```

①

**Note now that we also have the argument fun.**

②

**Returns the result of fun(first).**

Take some time to process what's intended to happen here: you have created a function that will call some other function (which you haven't defined). If this seems a little haphazard and lacking in rigor, well, that's understandable: you haven't added any

guarantees about the function `fun`, what arguments *it* needs, or what it will return. What if you try to evaluate the function without the second argument? You get an error:

```
first_letter_formatted(word = "dog")
#> [1] "dog starts with d"

#> Error: argument "fun" is missing, with no default
```

The function message was created and sent to the Console, but the rest failed when trying to evaluate `fun(first)`. That makes sense; you haven't told the function what `fun` should be.

You need a function that takes a string, does some formatting, and returns another string. There's nothing that says this in the code, but that's what's intended. You could make that clearer by adding some comments to that effect.

## Documenting functions

Good documentation makes it clear how a function *should* be used; it defines a *contract* between you, the author/user, and the function. You agree to pass in certain *types* of data, and the function agrees to process them in a certain way, returning a certain *type* of result.

Use comments to describe what the inputs and outputs *should* be, and what you intend the internal processing to accomplish. For example, this might be a trivial example:

```
## adds one to a number
## input: n, a number
## output: a number, n+1
add_one <- function(n) {
  return(n + 1)
}
```

**TIP** There are much more sophisticated ways to do this once you

start writing packages, including testing (`testthat`) of these contracts and automatic documentation of arguments (`roxygen`).

Some functions you could use are `toupper` and `tolower`, which convert strings to uppercase or lowercase, for example:

```
toupper(x = "text processing is fun")
#> [1] "TEXT PROCESSING IS FUN"
```

You can pass one of these to your function:

```
first_letter_formatted(word = "dog", fun = toupper)
#> [1] "dog starts with d"
#> [1] "D"
```

Note that you don't pass `toupper` as a string, and you also don't *call* the function (which you'd do with parentheses, as `toupper()`). You pass the actual function to `first_letter_formatted` so it's used internally as if defined there. You can look at what you passed in (the function itself) by evaluating it, again not including the parentheses (which would mean you were *calling* the function):

```
toupper      ①
#> function (x)
#> {
#>   if (!is.character(x))
#>     x <- as.character(x)
#>   .Internal(toupper(x)) ②
#> }
#> <bytecode: 0x2aeadb0>
#> <environment: namespace:base>
```

①

No parentheses — you're asking for the function code, not calling the function

②

The full implementation is in some internal C code, but you're sending this code as an argument.

You can pass in a different function easily:

```
first_letter_formatted(word = "DOG", fun = tolower)
#> [1] "DOG starts with D"
#> [1] "d"
```

The `first_letter_formatted` function can take any compatible function as the `fun` argument, as long as it can process a string when called as `fun(x)`. This flexibility makes R amazingly powerful because it can build incredibly complex functions out of smaller components, passing around functions and data as required.

### 4.1.5 DEFAULT ARGUMENTS

The previous example extracted the first letter of several words. If you extend those to include formatting, you'll begin to recognize some repeated code:

```
first_of_cat <- first_letter_formatted(word = "cat", fun
= toupper)
#> [1] "cat starts with c"

first_of_dog <- first_letter_formatted(word = "dog", fun
= toupper)
#> [1] "dog starts with d"

first_of_fish <- first_letter_formatted(word = "fish", fun
= toupper)
#> [1] "fish starts with f"
```

There's an old term in programming known as DRY (Don't Repeat Yourself), which encourages removing unnecessary repeated structures in favor of functions or better design. In this case, you may want to use `toupper` most of the time, and you can set this as the *default* argument. The argument `fun` will still be recognized if requested, but will take a particular value by default.

The way to instruct R to use a default argument value is to specify it when defining the function, as follows:

```
first_letter_formatted <- function(word, fun = toupper)
```

```
{
  ①

  ## extract the first letter of the word using substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(fun(first))
}
```

(1)

If this function is called without the argument fun, the default toupper is used.

Now you can use this function, with the formatting behavior, despite only providing one argument:

```
first_letter_formatted(word = "house")
#> [1] "house starts with h"
#> [1] "H"
```

If for any reason you don't want to use the default, specify the value to use:

```
first_letter_formatted(word = "HOUSE", fun = tolower)
#> [1] "HOUSE starts with H"
#> [1] "h"
```

You've seen this in action already — many R functions have default values, so you don't need to specify them. One of those is `print()`, which you have used to print an object to the Console:

```
print(x = "a")
#> [1] "a"
```

If, instead, you wanted to leave the quotes off, you could use the `quote` argument (which has a default value of `TRUE` for strings):

```
print(x = "a", quote = FALSE)
#> [1] a
```

If you are using RStudio, pressing Tab after typing a comma when evaluating a function should bring up a tooltip window of the possible arguments to that function. The definitions for each of these (and whether they have default values) are usually found in the help file for that function.<sup>6</sup>

---

<sup>6</sup> Recall that the help file for some function `foo` can be reached either by pressing F1 while your cursor is on the function name or by evaluating `?foo`.

**CAUTION** Be mindful of default argument values; they aren't always what you might expect them to be. A particularly painful one for many R users appears when importing data, in which case the default `stringsAsFactors = TRUE` performs conversion on any text data automatically, and usually contrary to what is desired.

## 4.1.6 ARGUMENT NAME MATCHING

Until now, I've tried to make sure I explicitly name input variables in function calls, such as this:

```
print(x = "abc")
#> [1] "abc"
```

But do you really need to use `x =` every time? For better or worse, the answer is no. R will gladly make some assumptions about the input variables you use in function calls. If you don't name any of the inputs, R will assume they're being provided in the order in which they appear in the function definition. This is known as *positional* indexing. When functions only have a single argument, this becomes trivial, and the name is easily dropped. When a function takes several arguments, though, if you aren't going to use their names, it's important to ensure that you input them in the correct order. You can use the preceding `print()` function just fine without specifying that the argument name is `x`; R will figure that out:

```
print("abc")
#> [1] "abc"
```

For example, to calculate a random value, you can use the `runif()` (random uniform value) function from the (by default, available) `stats` package. If you look up the help file for this function using `?runif`, you see it takes three arguments:

- `n` — The number of values to generate
- `min` — The minimum value that could be produced
- `max` — The maximum value that could be produced

This means you can request `n` random values between `min` and `max`. You can keep working with the same “random” variables consistently if you set the `seed` value using the `set.seed()` function before evaluating a random selection, which I’ve done in each case here so that the results are consistent.<sup>7</sup>

<sup>7</sup> This is good practice any time you are using random numbers. Some people have a favorite seed — for example, their birthday, 42, or 1337. If you’re evaluating all of your code in order, then the seed only needs to be set once. I have set it each time here so you can rerun the examples in any order and get the same results.

If you are being (correctly) verbose and writing out argument names, you can request four values between 1 and 2 from `runif()` with

```
set.seed(1)
runif(n = 4, min = 1, max = 2)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

If you’re confident in these arguments, you can drop their names (note, these are random, but you’ve asked for the *same* random selection, so they are the same values as those just obtained):

```
set.seed(1)
runif(4, 1, 2)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

Coming across this line with unnamed arguments 4, 1, and 2 in

the middle of a large script should raise some eyebrows, especially if the function being called is more obscure. It's difficult to read into what those numbers represent when they are passed by position and not their names.

An added benefit of using the argument names is that they can commonly be provided in any order:

```
set.seed(1)
runif(max = 2, n = 4, min = 1)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

The very large caveat to this is when functions utilize the dot-dot-dot ellipses (...). This is a special argument that captures additional arguments and passes them to an internal function call, so you can send these arguments “down the line”:

```
nestedFunction <- function(N, ...) {
  runif(n = N, ...)
}
```

In this example, the ... argument will capture any additional arguments other than N (since it has no other named arguments) and use them in the `runif()` call:

```
set.seed(1)
nestedFunction(N = 3, min = 1, max = 2)
#> [1] 1.265509 1.372124 1.572853
```

Note that the definition of `nestedFunction` doesn't specify `min` or `max` at all; it just allows for “other arguments,” which it will pass to `runif()`.

If you choose not to use these, then nothing is passed to the `runif()` call, and the default values are invoked (`min = 0, max = 1`):

```
set.seed(1)
nestedFunction(N = 3)
#> [1] 0.2655087 0.3721239 0.5728534
```

(Note that because you aren't using the arguments used previously, these aren't the same random variables previously

obtained). You don't even need to name these additional arguments; the first will be assumed to be the first positional argument, and the remaining arguments will be captured by . . . :

```
set.seed(1)
nestedFunction(3, 1, 2)
#> [1] 1.265509 1.372124 1.572853
```

The other useful place you'll commonly find this feature is when a function requires an indeterminate number of arguments, such as `sum()`, which calculates the sum of its arguments, no matter how many, without requiring a name for each value:

```
sum(12, 13, 9, 11)
#> [1] 45
```

## Similar but different

Be warned: different functions have different arrangements, names, and defaults for their arguments. Because `sum()` takes any number of arguments and calculates the sum of their values, you may expect `mean()` (calculate the average) to do something similar. Of course, I wouldn't be mentioning it if it did.

The help file for the `sum()` function (using `?sum`) shows that the first argument (and only unnamed argument) is . . . , which means "any number of variables not specifically named here":

```
sum(1, 2, 3, 4)
#> [1] 10
```

Try that with the `mean` function, which should take the average of the values:

```
mean(1, 2, 3, 4)
#> [1] 1
```

Something isn't right. The average of those values is 2.5. If you go back to the help file (`?mean`), it says `mean()` has an `x`

argument before the `...`, and the value description notes that the function calculates the mean of `x`. In this case, the function is expecting more than one value to be stored in `x`. You'll see how to do that when we talk about combining values into larger structures.

**TIP** You'll probably want to look at the help file for every function you use at least once. When written well, the help file contains a useful description of how and when to use a function, potential hazards to watch out for, and links to other similar or related functions. If you're using a function and not getting the answer you expect, the help file is the first place to check.

### 4.1.7 PARTIAL MATCHING

R does many things to try to help you get work done. For better or worse, many of these things involve quirks of the language. One is the fact that if you supply enough of an argument name that R can figure out which one you mean, then R considers that to be enough.

This behavior is called *partial matching* of arguments, and it can either be a blessing (rarely) or a curse (more often). Let's say you want to generate 10 random numbers, drawn from a normal distribution.<sup>8</sup> The function to generate these values is `rnorm()` and is available by default from the `stats` package. It has the following arguments:

---

<sup>8</sup> Knowing exactly what this means isn't critical; I'm just specifying how likely it is to produce each number. In this case you're more likely to produce a number close to the `mean` (average) value.

- `n` — The number of values to generate
- `mean` — The mean of the normal distribution (default 0)
- `sd` — The standard deviation of the normal distribution (default

1)

Calling this function generates random numbers with a higher probability of being near the mean (in this case, the default value of 0). Again, use the `set.seed()` function to reproducibly sample randomly. Let's select 10 values using `rnorm`:

```
set.seed(1)
rnorm(n = 10)
#> [1] -0.6264538  0.1836433 -0.8356286  1.5952808
#> [5]  0.3295078 -0.8204684
#> [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

The defaults for the arguments `mean` and `sd` are 0 and 1, respectively. If you want to call `rnorm()` with different arguments, you certainly can:

```
set.seed(1)
rnorm(n = 10, mean = 10, sd = 2)
#> [1] 8.747092 10.367287 8.328743 13.190562 10.659016
#> [5] 8.359063 10.974858
#> [8] 11.476649 11.151563 9.389223
```

But you can be lazy and only supply enough of the variable names such that R knows which ones you're talking about:

```
set.seed(1)
rnorm(n = 10, m = 10, s = 2)
#> [1] 8.747092 10.367287 8.328743 13.190562 10.659016
#> [5] 8.359063 10.974858
#> [8] 11.476649 11.151563 9.389223
```

Note that you didn't specify the `mean` and `sd` arguments, but `m` and `s` partially match to those uniquely, so R goes ahead and works with what it has. This works when the function is defined with named arguments (these are called *formal arguments*) rather than the catch-all ..., which doesn't specify what name the arguments must have.

As you can imagine, this isn't always so smooth. Often arguments will have many letters in common at the start of their names, so it's best to not rely on this behavior at all. For example, if instead of a normal distribution you wanted random

numbers drawn from the uniform distribution you saw in the previous section, you could call the `runif()` function. Calling this with fully named arguments works as expected:

```
set.seed(1)
runif(n = 10, min = 0, max = 1)
#> [1] 0.26550866 0.37212390 0.57285336 0.90820779
#> [5] 0.20168193 0.89838968
#> [7] 0.94467527 0.66079779 0.62911404 0.06178627
```

But trying to partially match with just a single character fails because it doesn't match uniquely to a single argument:

```
set.seed(1)
runif(n = 10, m = 5)

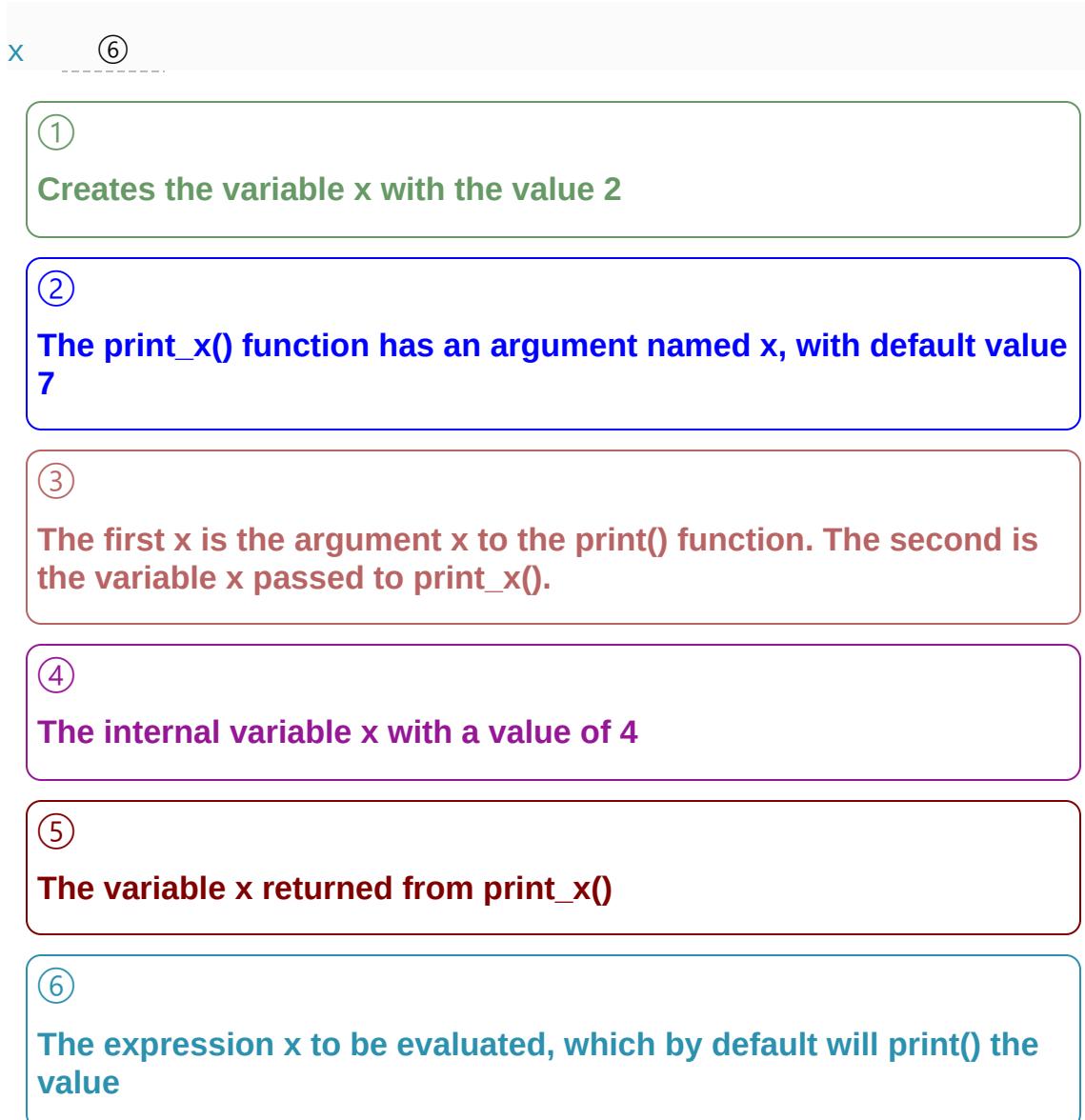
#> Error: argument 2 matches multiple formal arguments
```

It's best practice to always specify your argument names in full so as not to rely on this fragile feature, or to cause confusion with raw data being assigned to arguments by position alone.

### 4.1.8 SCOPE

It's important to understand what's happening to variables that are created and manipulated within a function. I've told you that once a variable is used as the value of an argument, it's left alone and unchanged by whatever happens inside the function, but what if you try to create an internal variable with that same name? One easy way to find out, of course, is to try!

```
x <- 2      ①
print_x <- function(x = 7) {      ②
  print(x = x)      ③
  x <- 4      ④
  return(x)      ⑤
}
print_x(x = x)
```



That's a lot of instances of `x` to keep track of. With trial and error, you might be able to figure out the rules that R will follow, but I'll tell you what they are, and then you can guess what the final evaluation will produce.

Most programming languages have some concept of *scope* — the parts of code for which a variable (or object in general) will be relevant. Consider for a moment the first names of everyone you know: Alice, Bob, Chris, ... This is the widest scope of “people names” you have. Now, imagine you move to another country away from everyone you know and begin to create a new

list of people names as you meet new people. This scope is *narrower* because it is more relevant to you right now, in your new home. Some of these new people might have the same first name as those you knew back home — the two Bobs are still different people, they just have the same label.

In this same way, while R is evaluating the instructions within a function, it creates new variables that may or may not have the same name as variables you defined outside of the function, but internally the memory addresses (or last names, to continue the analogy) of these are different. So, the internal variable `x` inside the `print_x()` function might have the same name as `x`, but it's a new variable specific to that function. If it helps, think of this as `x_2`. The label `x` is only for your understanding — R knows what it's doing. Variables defined within a function's scope are commonly referred to as *local* variables. You still use the assignment operator `<-` to store these, but they are stored (and exist) only within the scope of the function.

What happens to these variables once the function has completed? This is where the analogy to people in different countries breaks down. Whereas “local” Bob will probably live on after you return home, the local variables created inside a function do not. This is part of the *functional programming* paradigm; data should be unchanged, and functions shouldn't have *side effects*, which means that merely evaluating a function shouldn't change any data.

Can you guess what the preceding code will produce? You now know that the value `x` won't be changed by the function, even though the function temporarily creates another variable named `x`. Let's walk through the code and see. First assign the value 2 to `x`:

```
x <- 2
```

Now define the function:

```
print_x <- function(x = 7) {
  print(x = x)
  x <- 4
```

```

    return(x)
}

```

If you evaluate the function with the variable `x` given to the argument `x`, you see the following:

```

print_x(x = x)
#> [1] 2
#> [1] 4

```

The first result comes from the `print()` function, which prints the value of the local variable `x`. Because you don't assign the return value of the function to anything, it is also printed to the Console. If you now check what value `x` has

```

x
#> [1] 2

```

you see that it remains unchanged.

What if you evaluate just the new function without any arguments?

```

print_x()
#> [1] 7
#> [1] 4

```

In this case, because you didn't specify what the local version of `x` should be, it is assigned the default value, which is what the `print()` function receives. This is printed, and then this local variable is assigned the value 4, which is returned and thus printed. The original variable `x`, though, is still unchanged:

```

x
#> [1] 2

```

A useful solution to this is to name your variables appropriately. `x` is almost always a bad choice of name for data, but can make sense as an argument to a function if it is a completely general input.

The local variables are destroyed once the function evaluation is completed, which means you can't access them outside of that scope. If you try to, R will complain and return an error:

```
y_in_scope <- function(y) {
  print(x = y)
}

y_in_scope(y = 5)
#> [1] 5
```

Here the value of `y` only exists within the scope of the function, so you can't use it outside of that:

```
y

#> Error: object 'y' not found
```

This can be of great benefit — if a function requires several intermediate steps, then the intermediate values only exist for as long as the function is being evaluated. The inputs and outputs are all that remain once the function evaluation is completed, so you don't have unnecessary variables floating around.

R will search through the different levels of scope when it tries to find what value a variable might have (if it has one). It begins with the scope which the expression is in (a function, or the Workspace). If R can't find a variable with that name in that scope, it moves to wider and wider scopes. With functions inside functions, R moves up through the order in which they were called until it reaches the Workspace. If it fails to find the variable there, it searches the loaded and attached packages (see section 4.2 for a discussion of packages). Only when it fails to find the variable in any of these does it return the error.

This can be very useful, but also very dangerous. The preceding example showed that the variable `y` would not persist outside of its function's scope. The opposite scenario occurs, though, when a variable in the Workspace is *not defined* within a function. In that case, after failing to find the variable in the function's scope, R will look at the wider scope (the Workspace) to see if it can find the variable before returning an error:

```
does_not_contain_z <- function(x = 1) {
  print(x = z)
}
```

This is somewhat dangerous because in isolation, this function will fail. The scope of this function seems to know nothing about a `z` variable. Sure enough:

```
does_not_contain_z()
#> Error: object 'z' not found
```

But if the Workspace *does* contain a variable `z`, then R finds it when it expands its search to a wider scope:

```
z <- "apple"
does_not_contain_z()
#> [1] "apple"
```

The difference between this example and the previous with all the `x` variables is that `print_x()` *did* contain a variable `x`, so R created a new `x` variable within that function's scope. Here, R widens its search through the scopes until it finds the variable `z` (or doesn't).

In this way, you can define some variables that will *always* be available to functions you create, with the disclaimer that they won't be used if your functions define their own versions of that name. These are commonly referred to as *global variables* because they are always defined. Anything defined in your Workspace is a global variable because this scope is wider than that of any function you create.

**TIP** Using good variable names helps to avoid any conflicts — make your local variable names meaningful to their scope, and make sure you always define them with some value.

R will always begin with the most *narrow* scope, so you could prevent the previous scenario by defining a *local* `z` variable, even if it's meaningless:

```
z <- "apple"
does_contain_z <- function(x = 1) {
  z <- "nothing"
```

```

  print(x = z)
}
does_contain_z()
#> [1] "nothing"

```

It's important to note that R will only partially check that a function makes sense when you evaluate its definition. R will check that parentheses and curly braces are all matched up, and that the body of the function contains valid R code, but it won't check that variables or other functions are guaranteed to exist. This is due to the scoping we just discussed; variables and functions could very well be defined outside of the function you're defining in a wider scope, so there would be no way for R to check this when you define a new function.

This is a blessing when it comes to *recursion*, the concept of performing an operation that calls itself. You can define a function `factorial()` that calculates the product  $n \times (n-1) \times (n-2) \times \dots \times 1$ . To do this, either loop over the values of the input  $n$  from  $n$  down to 1, or take advantage of the fact that you need to essentially do the multiplication again with a smaller number many times. Stop when you get to 1, at which point you will return the final value. A function that does that looks like this:

```

## returns x*(x-1)*(x-2)*...*1
factorial <- function(x) {

  ## if the input is 1, just return the input
  if (x == 1) {
    return(x)
  } else {
    return(x * factorial(x - 1))
  }
}

```

Now, before you evaluate this function, R has no idea what the `factorial()` function looks like, but it *can* be defined, and that's good enough for R. As soon as you evaluate this function definition, there *is* a definition for `factorial()`, so the function can now be called. When you call it, R checks to make sure it

knows what it's looking for (according to the scoping rules discussed previously) and finds it, since you've defined it. R has no issue with calling it again, despite already being in the process of doing so, since it creates another narrower scope for the new call to `factorial(x - 1)`. It continues doing this until the recursion exits, which you've allowed for by checking when the input is 1.<sup>9</sup> You can check how the recursive function stacks up against doing the calculation manually, and also against the built-in version from `base`:

---

<sup>9</sup> It might help to know that the formal definition of a factorial function, often written as `x!`, defines the value `1! == 1`.

---

<sup>10</sup> `base::factorial()` is a good six times faster than the example version, partly because it's written in C.

```
c(factorial(3), 3*2*1, base::factorial(3))
#> [1] 6 6 6

c(factorial(6), 6*5*4*3*2*1, base::factorial(6))
#> [1] 720 720 720
```

Next we'll look at how to expand the already wide variety of functions available, with *packages*.

## 4.2 Packages

A default installation of R comes with a wide variety of functions to complete an astonishingly diverse range of tasks, bundled together in the form of *packages*. The most basic of these is the `base` package, which, along with the few predefined values such as `pi`, defines the basic structures of the language.

**DEFINITION** A *package* is a collection of functions and data that extends the functionality of the language.

You can most easily examine all of the `base` package functions by requesting a listing of them via the `help()` function:

```
help(package = "base")
```

You can also view the entire namespace using the Environment tab drop-down box (show the namespace of a specific package rather than the Global Environment). This package contains many of the functions you've been using so far, including `print()`, `as.integer()`, and `typeof()`.

Common summary functions are also found in the base package, such as `sum()`, `mean()`, `length()`, and `nchar()`. Additionally, several packages are bundled with R installations that provide some essential functionality. The `utils` package contains many functions for interacting with the wider system, such as reading files. The `stats` package contains functions for performing statistical tests, among other things. The `graphics` package provides basic plotting functionality, and the `datasets` package includes a large number of built-in datasets that, because they are bundled with all installations, make for good testing/example values.

Frequently, the preinstalled functions are not enough for the analysis you want to undertake. If you understand what you need to do and can write the function(s) yourself, then R is ready and waiting for you to do so. In many cases, though, this effort has already been exerted by someone else and published as an available package, either on the Comprehensive R Archive Network (CRAN, which implements a variety of tests to ensure that packages will install on various operating systems without issue), Bioconductor (another open source repository of R packages specific to the study of genomic data), or a version control repository such as GitHub (for which testing is undertaken by the author, if at all, but which has the advantage of wider community input and faster development schedules).

To date, there are roughly 12,000 packages available on CRAN. There's a good chance that if you need a function to perform a well-known task, a package already achieves that goal. The quality (in terms of freshness, accuracy, and ease of use) of said package is not necessarily guaranteed (though surprisingly

few packages on CRAN are broken), but with open source software there is always the opportunity to improve it. These packages are written by all levels of R users — from new users who fill a niche and create packages that may only be used a handful of times to full-time R developers who push the boundaries of the existing language and create wonderful new ways to interact with data.

A list of the packages available on CRAN can be found at <https://cran.r-project.org/web/packages/>, and these can be sorted by name or publication date. I find the easiest way to find a relevant package is to query a search engine with descriptive words related to what I'm trying to do, plus either *R* or *rstats*. A search like that commonly brings up a discussion board or Stack Overflow question/answer where a package will be recommended.

**DEFINITION** A *dependency* is an additionally required (separate) package that contains code/data to enable functionality within a package. A package may have many dependencies, and the R package framework allows those to be formally listed and installed when required.

When extending R by writing a package, you don't need to reinvent the wheel (or your own versions of all the functionality already available). Package authors can utilize functions from other packages, and when they do so that package becomes a dependency of their package. R keeps track of these as part of the package framework: when a package is installed, all the dependencies are also installed, so all of the functionality should be available.<sup>11</sup> For example, the `dplyr` package has the `Rcpp` package as a dependency, since `Rcpp` contains functions for interacting with the C++ code `dplyr` relies on. By allowing these dependencies, common functionality can be isolated into small packages that many other packages may then utilize.

---

<sup>11</sup> There are some exceptions to this, such as packages listed as `suggests`; these aren't strictly required, but do offer further functionality, such as in examples, vignettes, or rarely used functions.

---

## 4.2.1 INSTALLING PACKAGES

To install a package from CRAN by name, `install.packages()` takes a package name as the first argument, and this is typically sufficient (provided you've set up CRAN or a local mirror of CRAN as the repository to search — otherwise R will likely ask you). To install the popular plotting package `ggplot2`, enter the following:

```
install.packages("ggplot2") ①
```

①

The first argument `pkgs` requires a string — the name of the package to install.

What will be shown in the Console (if all goes well) is a summary of the steps taken in installing a package: typically downloading, unpacking, installing dependencies, building, testing, and finally installing the package. All of this happens automatically and rarely requires intervention. If this succeeds, you will be presented with something like the following:

```
* DONE (ggplot2)
```

```
The downloaded source packages are in  
  '/tmp/RtmpTf0kV8/downloaded_packages'
```

Keep in mind, though, that installing a package only makes it available for *loading* (reading the function definitions into memory) and *attaching* (adding the function names onto the search list so that they can be looked up). You still need to *load* and *attach* the package before the functions can be used easily, which is best performed using the `library()` function:

```
library(ggplot2) ①
```

(1)

**The first argument package can be a string (the name of the package to load and attach) but can also be a bare name (without quotes, as shown here) of the same.**

To *detach* a package (should you decide you really need to), it's easiest to restart your session (RStudio shortcut: Ctrl-Shift-F10). There are functions to facilitate detaching a single package, but these require delicate handling to ensure that the dependencies are (or are not) also removed.

## A common package installation issue

When trying to install a package, it's common to have R complain that a package is not available for your version of R:

```
install.packages("bloopblorp")
# Installing package into '/home/user/your/R/library'
#> (as 'lib' is unspecified)
# Warning message:
# package 'bloopblorp' is not available (for R version
3.4.3)
```

Sometimes this is because you've mis-entered the name of the package you want (for example, `ggplot` instead of `ggplot2`). Other times it's because the package explicitly requires a newer version of R than you have. This is partly due to CRAN requiring that new packages be built using the latest version of R, and CRAN enforces the same requirement on the users. In any case, it's a good idea to remain up-to-date with the latest stable version of R, which may resolve this issue.

Searching an archive of Stack Overflow R-tag questions (189,930 questions, spanning September 2008 to October 2017: see [www.kaggle.com/stackoverflow/rquestions](http://www.kaggle.com/stackoverflow/rquestions)), I see 312 matches for the phrase “is not available (for R

version”, which means this isn’t a rare occurrence.

Once the package has been installed, the `install.packages()` step isn’t required again until you update the package to a newer version. The next time you want to utilize some functionality from an installed package, the `library()` function (with the argument of the package name) is all that’s required. With that said, though, the `library()` call *will* be required each time you start a new session (for example, exit and reopen RStudio). This can be simplified by loading certain packages at the start of any new session, or by grouping together those `library()` calls at the start of your scripts.

**CAUTION** When upgrading a package to a newer version, be very careful about reading the *changelog*, which details any significant changes. New versions aren’t always about fixing bugs — they can include improvements that involve changes to the way functions are called (different arguments, different defaults), and these can have undesired consequences to your code.

If you want to work with “in-development” packages not hosted on CRAN (such as GitHub), you can, but it’s a good idea to understand the trade-offs involved. Packages hosted on CRAN have undergone a detailed testing period that ensures they meet certain criteria (have documentation, pass any consistency tests, install correctly, refer to dependencies correctly, and many more). Packages on GitHub are in whatever state that particular developer has them in, with no guarantees of what the package does. That said, it’s a risk many people are happy to take, and as far as I am aware there aren’t well-known examples of things going horribly wrong.

To install a package from GitHub, first you’ll require the ability to build packages from their raw source (*binary* packages are

often available from CRAN that are already compiled and ready to use). The state-of-the-art package for R developers is `devtools`, and you can read more about how to install its dependencies (which may include RTools for Windows or Xcode for Mac) at <https://github.com/r-lib/devtools>. Once you have that installed (using `install.packages("devtools")`) and loaded (`library(devtools)`), you can install packages from external sources. To install the package hosted at <https://github.com/username/repository>, do the following:

```
# install.packages("devtools")      ①
library(devtools)                 ②
install_github("username/repository") ③
```

①

**Installs the `devtools` package from CRAN. Quotes around the package name are necessary here.**

②

**Loads and attaches the `devtools` package. Quotes around the package name aren't necessary here.**

③

**Note the name of this function does not involve a dot, but rather an underscore.**

Whenever I show code that requires an extension package, I will add the `install.packages("package")` and `library(package)` lines with the `install` step commented out; you only need to install packages once (until you upgrade them). If you haven't already installed the package, you will need to uncomment that line (or type it into the Console yourself) and evaluate it.

## Package scopes for functions

There's no guarantee that two packages won't use the

same name for a function. To specify exactly which package's scope to take a function (or variable) from, prepend the name with the package and two colons. This applies when multiple packages provide functions with the same name, or you write a function and use the same name as a package's function. For example, if you wrote a `sum()` function, even though there is already one available in `base`, you can still use both as long as you're specific about requesting the one from `base` with the package prefixed:

```
# A bad 'sum' function which just returns 0
sum <- function(...) {
  return(0)
}
sum(1, 2, 3)
#> [1] 0
# Explicitly use the 'base' version
base::sum(1, 2, 3)
#> [1] 6
```

This can also be useful if you don't want to load an entire package just to use a single function. For example, you may want to use `install_github()` from `devtools` just once, and you can do so like this:

```
devtools::install_github("username/repository")
```

## **4.2.2 HOW DOES R (NOT) KNOW ABOUT THIS FUNCTION?**

One of the most commonly encountered errors for those new to R is finding that a function doesn't appear to be defined:

```
someFunction()

#> Error: could not find function "someFunction"
```

Searching the Stack Overflow data archive (mentioned in the preceding section) for the preceding error message ("could not

“find function”) produces 942 results — 0.5% of the questions asked included that error message in the question body. So if you encounter this error, relax, you’re not alone.

This occurs when R can’t find any definition for a function that appears in the expression to be evaluated. To understand how R *doesn’t* know about some functions, you first need to understand how R *does* know about other functions.

In the scope discussion, you saw that R will look through wider and wider scopes to find a value for a variable you ask for, starting within functions and expanding out. Functions are searched for similarly. If you define a function in your Workspace, the code that it contains is stored in the *global namespace*. A *namespace* is merely the list of names of things (variables, functions, packages) within a given scope.

Just like variables, if you define a function *within* another function, then that defines the limit of its scope, and the function can’t be called on elsewhere. When you ask R to evaluate a function, it searches the global namespace for any functions defined there. If it can’t find one, it expands the search to packages in the search path.

**NOTE** Packages that appear in the search path, and that therefore will be found when scope-searching, are those that you have *installed*, *loaded*, and *attached* using an installation function and the `library()` function. *Attaching* a package means the contents of the package are finally added to the search path, and so functions are available for you to use. Merely installing a package isn’t enough for it to be available so easily.

This means there is a hierarchy to the namespace. In a hypothetical situation where you attach `packageA` containing a function `blorp()`, then attach `packageB` containing a different version of `blorp()`, and then define your own version of `blorp()`, R will *mask* the function each time (hopefully with a warning).

Calling the function will take the definition from the narrowest scope: in this case, the definition within the global namespace. To use a function from an installed package without loading the entire namespace (for example, to prevent masking), specify that using a double colon (::), which will tell R where to search for a function — for example, packageA::blorp().

### 4.2.3 NAMESPACES

Another name for a namespace is an *environment*. You've seen that defining variables in the global environment causes them to appear in the Environment tab in the Workspace pane of RStudio. You may notice a drop-down box at the top of this pane that by default lists the Global Environment. Clicking this lets you select a different environment to list, and lists the currently loaded and attached packages. Selecting one of these will show the namespace for that package — typically a long list of functions.<sup>12</sup>

---

<sup>12</sup> These will commonly show up as <Promise> by virtue of the way that R *lazy loads* (delays formally defining) objects it doesn't necessarily need until they are used. If you start typing the name of one of these in the Console, RStudio's code-checking tools will likely be enough of a trigger to change the <Promise> to a full-fledged function.

Back to the original question: what is wrong when R “could not find function ‘someFunction’”? Hopefully, now you see that R will have searched the global namespace and all loaded and attached package namespaces and not found a definition matching someFunction(). That means either the package containing that function hasn't been installed, or merely hasn't been loaded.

The first thing to check is whether it has been loaded, or, failing that, whether it has been installed at all. You can list all of the loaded namespaces with the function `loadedNamespaces`:

```
loadedNamespaces()
#> [1] "compiler"   "magrittr"   "graphics"   "tools"
"utils"
```

```
#> [6] "switchr"     "grDevices"   "stats"       "datasets"
"stringi"
#> [11] "knitr"       "methods"     "stringr"     "base"
"evaluate"
```

(Or look through the list in the Environment pane's drop-down list.<sup>13</sup>) If it isn't there, you can check if it is installed. The simplest way is to view the Packages tab in the same pane as Help. If it's not listed there, then the package hasn't been installed. For now, there's plenty to see in the packages that come with R by default, but you won't break anything by installing another package and not using it.

---

<sup>13</sup> Note: these may very well be different from what you get if you try this in your session, depending on what libraries you've used in your session so far.

## Masking of objects

You've now seen what happens when R can't find a function, but what about when it *can* and you try to use a package that also has a function with that name? For example, say you've written a `count()` function that returns the number of characters in a string (this is already a function — `nchar()` from the `base` package):

```
count <- function(string) {
  nchar(string)
}
count("abcde")
#> [1] 5
```

Then try to use the `dplyr` package, which also contains a `count()` function (with a much more useful implementation):

```
# install.packages("dplyr")
library(dplyr)
#
#> Attaching package: 'dplyr'
#> The following object is masked _by_ '.GlobalEnv':
#>
#>     count
```

```
# The following objects are masked from
'package:stats':
#>
#>     filter, lag
# The following objects are masked from
'package:base':
#>
#>     intersect, setdiff, setequal, union
```

The `count()` function takes precedence because it is in the global environment (The following object is masked by `'.GlobalEnv': count`). Some other functions are masked from other packages because `dplyr` is now higher on the search path (the more recently a package is loaded and attached, the higher its precedence).

## 4.3 Messages, warnings, and errors, oh my!

It's inevitable that sooner or later something won't work, and R will tell you about it. This will appear as text in the Console that does not begin with a prompt (>) and may be colored differently from your normal input command text.

R has several different types of information it can send to you regarding the most recent evaluation:

- *Messages* — This doesn't mean anything went wrong, but the developer who wrote the function that raised the message allowed for R to tell you something — perhaps some additional information to go along with your data result, or a notice that the function will behave differently in future versions. These will be printed to the Console immediately and won't stop your commands (or script) from running, but don't become complacent and ignore them either; messages about masking of functions can be very important.
- *Warnings* — Something, perhaps unexpected, has happened that is not critical but is also not desired. This can be as innocent as `NA` values being introduced, or something as critical as your regression model not converging but exiting gracefully

nonetheless. R will keep track of accumulated warnings, and once your commands (or script) complete, it will print them to the Console.

- *Errors* — These occur when something has failed entirely. These will usually cause your commands (or script) to exit prematurely and require careful inspection to diagnose. These will be printed to the Console immediately, and the scope will revert back to the global environment, making an error arising from within a function call somewhat difficult to diagnose (because any local variables will be destroyed by the time you regain control of the session).

### **4.3.1 CREATING MESSAGES, WARNINGS, AND ERRORS**

Follow along with these examples and observe the different outputs.

#### **MESSAGES**

You can create a function that produces a message. The `message()` function takes any number of objects that it can convert to text and paste together. A single string is also quite useful:

```
raiseMessage <- function() {
  message("Keep going, you'll be a user in no time!")
}
```

Call this function to observe the effect it has:

```
raiseMessage()
# Keep going, you'll be a user in no time!
```

Note that there isn't any context provided — the message is merely printed to the Console. This also doesn't start with a [1] because it's not a returned value.

## WARNINGS

Warnings behave differently. These are queued until the running expression is complete, at which point a summary is produced. If 10 or fewer warnings are produced, they are printed to the Console. If more than 10 are produced, then the number of warnings is noted with the instruction to use the `warnings()` function to view them all (the first 50 at least). If you accumulate at least 50 warnings (which can be surprisingly easy once you learn how to repeat operations), R will present a message stating the following:

```
There were 50 or more warnings (use warnings() to see the
first 50)
```

As it suggests, you can list these using the `warnings()` function.

You can create a function to produce a warning:

```
raisewarning <- function() {
  warning("Ouch! Please don't do that.")
}
```

Evaluating this function, you may notice a different color in the output (depending on your theme settings):

```
raisewarning()
# Warning message:
# In raisewarning(): Ouch! Please don't do that.
```

Here you have some context that the text relates to a warning. It's a good idea to check the Console on completion of any evaluations.

## ERRORS

Although you're unlikely to want to terminate your functions arbitrarily, you can create a function that does that. Unlike the other two commands, this one does not follow the same naming scheme (in which case this function would be called `error`).

Instead, the `stop()` function performs a termination:<sup>14</sup>

---

<sup>14</sup> This is because the `stop` function does a lot more than terminate the code execution; R can *catch* these errors and handle them in a more sophisticated way than merely terminating the execution, but that's beyond the scope of this book.

---

```
raiseError <- function() {
  stop("I'm sorry, Dave, I'm afraid I can't do that")
}
```

You can evaluate this (albeit briefly) in the middle of several operations:

```
a <- 2; raiseError(); b <- 3
#> Error: I'm sorry, Dave, I'm afraid I can't do that
```

Recall from section 1.4.1 that commands separated by a semicolon can be placed on the same line. And note that the variable `b` is never assigned its value; check the Environment pane and observe that `a` is listed but `b` isn't — the execution was terminated as soon as the error was encountered. This can make life difficult when an error is encountered in the middle of a large block of processing, because it becomes unclear whether certain pieces of code have been evaluated. When encountering an error, it's often a good idea to repair the issue and then start fresh with a new session.

`stop()` can be used within functions or scripts when you really don't want the code to execute any further if a condition is met. In this case, another function is available that performs a test before exiting:

```
smallValueCheck <- function(x) {
  stopifnot(x < 10) ①
  message("Look, the function made it this far!")
}
```

(1)

< is the less-than operator.

You can evaluate this with a “large” value (less than 10 is the condition required to avoid the error; execution will stop if the condition is *not* true):

```
smallValueCheck(20)
#> Error: x < 10 is not TRUE
```

But if you try with a “small” value, all is well:

```
smallValueCheck(5)
# Look, the function made it this far!
```

Notice that no specific error message can be provided to the `stopifnot()` function itself. The help file for this (via `?stopifnot()`) shows that *all* of the arguments must be `TRUE` (not `NA`) to avoid terminating code execution. The statement that is output will indicate the *first* argument that was not `TRUE`; as many tests as you require can be separated by commas in the call to `stopifnot()`.

You’ll inevitably have messages, warnings, and errors show up at some time or other. Next, you’ll learn how to deal with them.

### **4.3.2 DIAGNOSING MESSAGES, WARNINGS, AND ERRORS**

Messages aren’t of any concern, but they can be distracting if they show up too often or unsolicited. One common scenario in which they’re found is when you load and attach a package using `library()` when the new package masks objects already defined. You saw this earlier in section 4.2, when loading and attaching the `dplyr` package, which looks like the following (if you’re starting with a clean session):

```
library(dplyr)
#
#> Attaching package: 'dplyr'
```

```
# The following objects are masked from 'package:stats':
#>
#>     filter, lag
# The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

Here you're provided with some potentially helpful information, that the namespace for the package `dplyr` has taken precedence in the search list over some other namespaces (recall section 4.1.8). This can become annoying if you are already familiar with it and don't need to be reminded. Although any message can potentially change in the future, and you should really pay attention to them, they can be suppressed using the `suppressPackageStartupMessages()` function:

```
suppressPackageStartupMessages(library(dplyr))
```

(1)

**Produces no output (no messages)**

General messages can be suppressed using the `suppressMessages()` function. Because they're printed to the Console, there isn't much else you need to do with them.

If more than 10 warnings are produced, use the `warnings()` function to view a list of them. The warning text will be printed to the Console, and will hopefully provide sufficient information to diagnose what went wrong. To suppress these, use `suppressWarnings()`.

Errors are usually a showstopper, so it is unsurprising that there's a more in-depth way to deal with them. If your code has stopped executing because of an error, request a *trace* of what R was trying to do (which functions it was calling) just before it reached that point. The `traceback()` function prints the *call stack*: the expressions R was evaluating, possibly their line numbers relative to their source, and some of the values in use. The benefit of this is highly variable; sometimes it points to a malformed expression in an easily identifiable chunk of code.

Other times it merely unpacks the function calls without really giving any indication of why the code didn't work. The output of  `traceback()` is the sequence of calls (in reverse order, such that the call you made comes at the bottom) that led to the error. This will likely involve many functions you're unfamiliar with, mostly the internal structure of the functions you use.

An example of this might be when you passed a large value to the  `smallValueCheck()` function defined earlier:

```
smallValueCheck(100)
#> Error: x < 10 is not TRUE
```

You can examine where this error arose more specifically:

```
traceback()
#> 3: stop(msg, call. = FALSE, domain = NA)      ①
#> 2: stopifnot(x < 10) at #3                   ②
#> 1: smallValueCheck(100)                      ③
```

①

**The cause of the error, a `stop()` call within `stopifnot()`**

②

**The internal step through which the error can be traced**

③

**The call you evaluated**

In each case, you can use the  `debugonce()` function to enlist the help of a *debugger*.<sup>15</sup> Providing the name of the function as an argument to  `debugonce()` tells R to not merely execute the function body, but allow you to step through the lines at your own pace and try to learn exactly what is happening. To *debug* the  `areacircle()` function defined earlier, pass this name to  `debugonce()`, which flags it for debugging the next time it is evaluated:

---

<sup>15</sup> Not nearly as effective a tool as the name might suggest, but nonetheless very useful, a debugger won't remove the bugs but will help you isolate them.

---

```
debugonce(fun = "areaCircle")
```

If you now evaluate `areacircle()` with some value, you're presented with a new prompt, this time starting with `Browse`:

```
areaCircle(3)
```

```
debugging in: areaCircle(3)
debug at #1: {
  area <- pi * radius^2
  return(area)
}
```

```
Browse[2]>
```

This identifies that you're now in a *debug* mode. A highlighted line notes the next line that will be evaluated. Press Enter or N (for *next*) to step through the lines of the body of the `areacircle()` function. Try this, and note that the Environment pane now lists only the variables defined in this narrower scope (it also lists `areacircle()` as the name of the environment).

To exit the debug mode, press Q or click the Stop button, at which point you will be returned to the Global Environment and your normal Console.

---

<sup>16</sup> Pressing Esc may also exit debug mode.

---

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Edsger W. Dijkstra, computer scientist and Turing Award recipient

Now you have your functions, and you know what to do if R complains about them, but do they work?

## 4.4 Testing

If you or I were building a machine and finally finished tightening up the last screw, we wouldn't just assume it would work the way we expect, would we? We might test that it behaves when we start it up the way we designed it to, feeding it some expected input and making sure we get the expected output, and no parts fall off or inputs get jammed. In the same way, we wouldn't necessarily assume that our functions work before using them for an analysis, would we?

When a big company makes a machine, it tests to make sure that the machine works as expected. A good company will also test to see what happens when the machine is used in other, perhaps less conventional ways. A company often tests that the safety mechanisms work when situations get dangerous, or pushes the machine to its breaking point to identify where that is. The same principles apply for building function. It's time to see how quickly they break (or fail to).

The `first_letter_formatted()` function worked well for the values I showed, but how flexible is it? Let's try some stranger inputs:

```
first_letter_formatted("3.14159")
#> [1] "3.14159 starts with 3"
#> [1] "3"
```

Okay, that worked surprisingly well. How about a number, not a string?

```
first_letter_formatted(3.14159)
#> [1] "3.14159 starts with 3"
#> [1] "3"
```

Not bad. What if it's a variable that stores a number?

```
first_letter_formatted(pi)
#> [1] "3.14159265358979 starts with 3"
#> [1] "3"
```

Impressive. What about a number that doesn't start with 0 through 9 (a negative number)?

```
first_letter_formatted(-7)
```

```
#> [1] "-7 starts with -"
#> [1] "-"
```

The reason why all these work is that the function calls `substring()` on the input argument `word`. Examining the internal code of that function shows that the first thing it does is convert its input into a character string using `as.character(text)`. This almost always works, so the function is fairly robust against odd inputs.

What about the `areacircle()` function?

```
areaCircle("blue")
#> Error: non-numeric argument to binary operator
```

What about something that *looks like* a number?

```
areaCircle("5")
#> Error: non-numeric argument to binary operator
```

The reason is that the `exponentiate` function (`^`, raise to a power) can't handle non-numeric (for example, character) arguments, unsurprisingly. If we needed it to be able to deal with this scenario without breaking, we would have to add some logic to ensure that `^` always receives a numeric value. We'll return to this in chapter 8.

Testing is a powerful way to ensure that functions work. Any time you create a function, it's a good idea to test it to make sure it's doing what you expect. We'll return to how to formalize this in section 10.2.1 on unit testing.

## **4.5 Project: Generalizing a function**

Your boss saw your `first_letter_formatted()` function and thought it was great! Now you need it to be able to extract the first two letters, but only sometimes. You could do either of the following:

- Write a new function that extracts the first two letters, and use

whichever you need at the time.

- Add an argument to `first_letter_formatted()` that specifies how many letters to format.

Hopefully by now you recognize that the second option is superior. It reduces the amount of code that is duplicated, which means fewer places for bugs to hide and fewer things to keep up to date.

Your task: rewrite `first_letter_formatted()` so that a new argument `nLetters` can be provided (with a sensible default). The value of `nLetters` should be passed to the last argument of `substring()` so that it extracts the letters in positions 1 to `nLetters`. Don't forget to rename this new, more-flexible function since it no longer extracts *just* the first letter all the time. Document what your function does, what *types* it expects the arguments to take, and what the expected output of two test cases should be.

When you're done, ask a friend to read through your new function definition and guess what the resulting object will look like given some inputs.

## 4.6 Try it yourself

In chapter 3, you manually converted between Celsius and Fahrenheit temperatures, but now you know you can do that routinely with functions, so let's give it a go:

```
# Convert Celsius to Fahrenheit ①
C_to_F <- function(temp_C) { ②
  temp_F <- (9L * temp_C / 5L) + 32L ③
  return(temp_F)
}

# Convert Fahrenheit to Celsius
F_to_C <- function(temp_F) {
  temp_C <- 5L * (temp_F - 32L) / 9L ④
  return(temp_C) ⑤
}
```

①

**Don't forget to add helpful comments.**

②

**Giving arguments meaningful names can be extremely helpful.**

③

**Giving internal variables meaningful names can save time trying to figure out what you meant to do.**

④

**Write calculations as clearly as possible. You'll thank yourself when you go to change them later.**

⑤

**Explicitly returning a value helps ensure you're returning what you think you are. Because `temp_c` is assigned on the previous line, if you left it with no implicit return statement, this function would print nothing when called.**

You can now use these functions to convert *any* numeric temperature values:

```
C_to_F(temp_C = 38)  
#> [1] 100.4
```

The advantage of naming the variable along with its units is that incorrectly specifying one generates an error. In this next case, I have tried to use the argument `temp_c` in the function that is defined with an argument `temp_F`:

```
F_to_C(temp_C = -40)  
#> Error: unused argument (temp_C = -40)
```

The correct argument leads to the function evaluating correctly:

```
F_to_C(temp_F = -40)
```

```
#> [1] -40
```

The two scales produce the same value at  $-40^{\circ}$ .

## ***Terminology***

- *Return value* — The object that is produced by a function, typically provided by a `return()` function, and which can be assigned to another variable
- *Abstraction* — Representing some idea or set of operations in a more general notation, such as a function with a nondescriptive argument
- *Scope* — The area of code for which a variable is relevant, such as a function, Workspace, or package
- *Namespace* — The list of names of variables/objects within a given scope
- *Side effects* — Any changes that persist outside of a scope, such as production of graphics, updating of data, or modification of the system, generally against the “functional” nature of R
- *Global variable* — A variable defined in the Workspace, therefore in the widest available scope, and therefore available to any functions defined there also
- *Local variable* — A variable defined in a narrower scope, such as in a function, which cannot be accessed from a wider scope
- *Debug* — To step through the code defining a function at the level of its scope to determine the cause of errors

## ***Summary***

- Functions can be stored in variables just the same as any other data.
- Abstracting expressions into functions means you can reuse them.

- Functions don't alter data — the returned value needs to be assigned.
- Function arguments are lazily evaluated.
- Documenting a function outlines what *should* happen when it is used.
- Functions can pass other functions in as arguments.
- Arguments can be provided either by name or position.
- The ... argument captures all non-formal arguments and passes them on.
- Argument names will be partially matched if they are uniquely identifiable.
- Packages extend the language by providing more functions.
- R will search through wider and wider scopes until it finds a requested variable (or doesn't).
- Every operation in R is a function call, even ones that don't look like functions (for example, +).
- Default arguments can be a blessing and a curse. Know what they are (use the `help()` menu).
- Keep an eye out for messages, warnings, and errors. These don't always have the most informative outputs, but they can at least help you search for what has gone wrong.

# 5

## *Combining data values*

---

### This chapter covers

- Creating structures containing grouped data: vectors, lists, data, frames, and more
- How these interact with missing data
- Inspecting and querying these new structures

So far we've only dealt with single-valued variables, but it's highly unlikely that your data looks so simple. You need a way to combine values into groups of data that together represent some larger concept. There are many ways to do this in R, and the way you store collections of data can impact how you interact with it later, so let's see some of the different ways to combine data.

### **5.1 Simple collections**

The simplest way to group together values is with the function `c()`. Feel free to refer to this function however you like, but the words *concatenate*, *combine*, and *collect* are all good options.

**CAUTION** Because `c()` is a function, it's a good idea to avoid naming your variables `c`, which may be tempting if you start with `a` and `b`. Granted, R is very clever at figuring out if you mean to use a variable or function, but having `c(c,c)` in your code just

makes for a confused reader. Using "c" (the character) as a value is probably okay, because you won't refer to it without the quotes.<sup>1</sup>

---

<sup>1</sup> With the exception of functions from some packages, such as `ggplot2`, that use Non-Standard Evaluation (NSE) in their arguments, which doesn't require the quotes. That's a whole other story.

Together, multiple values form a *vector*, made up of *elements*. You can create a vector of any basic (*atomic*) type of value with one important rule: all elements *must* be of the same type — numeric, integer, character, date, and so on. `c()` combines all of its arguments into a single vector:

```
c(1.2, 2.8, 3.5)
#> [1] 1.2 2.8 3.5
```

```
c("cat", "dog", "fish")
#> [1] "cat"  "dog"  "fish"
```

```
c(TRUE, FALSE, TRUE)
#> [1] TRUE FALSE TRUE
```

Even a single value can be represented in a vector:

```
c(6L)
#> [1] 6
```

This is the same output you receive when you evaluate `6L` on its own. In fact, it's impossible to create a literal *single* value without it being in a vector.<sup>2</sup> Now you see why R output always starts with a `[1]` — any non-zero number of elements forms a vector, even if it contains a single value.

---

<sup>2</sup> R has no “scalar” types; even if you think you have one, you don’t.

The spreadsheet equivalent of a vector is a selection of one or more cells. Usually these are all in the same row or column and can be referred to with a *range*, for example `A1:A8`. Spreadsheet software will also typically warn you if values in a column of a

table are of different types, though it will allow them (and pretend to work with them). An example of selecting a group of values in a spreadsheet is shown in [figure 5.1](#).

	A	B	C
1	1.2	cat	TRUE
2	2.8	dog	FALSE
3	3.5	fish	TRUE

**Figure 5.1** A selection of values in a spreadsheet.

If you try to create an empty vector, R treats this as `NULL` because it doesn't have a type to infer:

```
c()
#> NULL
```

You can explicitly create an empty vector of a specific type using a type-creation function:

```
numeric(length = 0)
#> numeric(0)
```

```
character(length = 0)
#> character(0)
```

```
logical(length = 0)
#> logical(0)
```

The arguments to `c()` need not be single values themselves; you can create a vector by combining shorter vectors just as easily:

```
c(1, c(8, 9), 4)
#> [1] 1 8 9 4
```

Some of the built-in datasets are simple vectors for convenience, such as `letters` (the lowercase English alphabet) and `LETTERS` (uppercase of the same).

If you try to print more elements than there is room for in one line of the Console, R breaks the line and continues on the next, showing the index of the vector at that point:

```
print(LETTERS)
#> [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
```

```
"N" "O" "P" "Q"
#> [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

The `str()` function provides some useful information about an entire vector, listing the *type* of the elements (which must all be the same) and the indices of the elements:

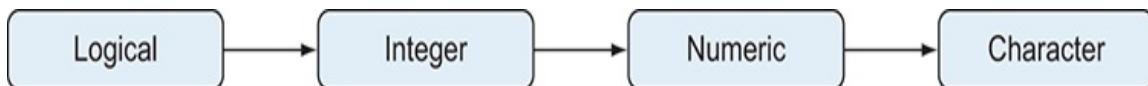
```
str(c(1.2, 2.8, 3.5))
#> num [1:3] 1.2 2.8 3.5
```

```
str(c("cat", "dog", "fish"))
#> chr [1:3] "cat" "dog" "fish"
```

```
str(c(TRUE, FALSE, TRUE))
#> logi [1:3] TRUE FALSE TRUE
```

### 5.1.1 COERCION

If you try to create a vector with a combination of types, R performs automatic coercion and ensures that they *are* of the same type by coercing to the most general type. Recall from section 3.5 that the ordering from least general (most specific) to most general is as shown in figure 5.2.



**Figure 5.2 Ordering of coercion**

```
c(2L, 4.3, 0) ①
#> [1] 2.0 4.3 0.0
```

```
c(TRUE, 5.3, 3L) ②
#> [1] 1.0 5.3 3.0
```

```
c(2L, 3L, "4") ③
#> [1] "2" "3" "4"
```

①

Coerced to numeric

②

Coerced to numeric

(3)

### Coerced to character

The *type* of a vector created with `c()` is therefore the coerced *type* of its elements, which will be the most general of the types involved:

```
typeof(c(2L, 3L, "4"))
#> [1] "character"
```

A vector created by combining smaller vectors, even if they have undergone coercion, will still be coerced to the most general type:

```
c("1", 2L, 3L, c(4.0, 5.1))
#> [1] "1"    "2"    "3"    "4"    "5.1"
```

## 5.1.2 MISSING VALUES

Missing values are allowed because they too have individual types: `NA_real_`, `NA_integer_`, `NA_character_`, and `NA` (the default is `logical`). These can appear in vectors without issue:

```
c(3L, NA, 5L)
#> [1] 3 NA 5
```

This is extremely useful for denoting when a value could have existed but doesn't. If you specify a particular flavor of `NA`, then that will also be involved in the automatic coercion:

```
c(2, 3, NA_character_, 5)
#> [1] "2" "3" NA "5"
```

The default `NA` is of type `logical`, so it is usually coerced when used in a vector of non-logicals, but it is printed as `NA` so it's not always clear that this has occurred.

The empty value `NULL`, though, cannot appear in a vector:

```
c(3L, NULL, 5L)
#> [1] 3 5
```

R will skip over this element entirely.

## Blank cells

Excel has no issue with leaving cells blank and will even skip over them in certain calculations. In R, there is no exact equivalent for a blank element in a vector, but you can use the missing-value placeholder `NA`. In a character vector, you could include the empty character string "", as in the following:

```
c("one", "two", "", "four")
#> [1] "one"  "two"  ""     "four"
```

But this isn't really *missing* as such, because ""*is* a valid value.

### 5.1.3 ATTRIBUTES

A vector of data may consist of thousands of individual values, and you can store those in a variable named with some care. You may be able to encode *some* information about the values in that name (perhaps `heights_of_giraffes`) but often you need a way to attach another label to the data.

R has a mechanism for this that works well when used internally, but it's somewhat fragile and I don't recommend you rely on it for anything you create yourself unless you fully understand the consequences. The mechanism is the attachment of *attributes* to objects (any object other than `NULL`) where additional information can be stored. For example, you might have a sequence of integers that represent something very important. You can store this sequence and refer to it with the variable `x`. The `attributes()` function lists any attributes of an object, and this new variable `x` has none:

```
x <- c(1, 2, 3, 4, 5)
attributes(x)
```

```
#> NULL
```

You can attach an attribute with the label "important thing" to `x` and provide the string value "critical value" with the attribute assignment (there is a function, `attr<-`, to which R translates this operation):

```
attr(x, "important thing") <- "critical value"
attributes(x)
#> $`important thing`
#> [1] "critical value"

x
#> [1] 1 2 3 4 5
#> attr(,"important thing")
#> [1] "critical value"
```

Some functions keep track of what these attributes are and pass them on to new objects:

```
y <- x + 1
attributes(y)
#> $`important thing`
#> [1] "critical value"

y
#> [1] 2 3 4 5 6
#> attr(,"important thing")
#> [1] "critical value"
```

Other functions do not (hence the danger in using this feature):

```
z <- c(y, 12)
attributes(z)
#> NULL

z
#> [1] 2 3 4 5 6 12
```

There are valid uses for attributes, but most of them are handled by specific functions, so it is rare that you'll need to interact with them this way. Knowing they are there, however, can be vitally important, as you'll see.

## 5.1.4 NAMES

Storing data as a collection is very useful when all the data represents a single unified concept, such as heights, counts, times, or cities. Often, though, these values have another layer of definition that you would like to retain to identify which value goes where.

Fortunately, elements of vectors in R can have individual names associated with them, so you can recall which value represents which thing. Specify these as argument names to `c()`, which will create a vector from all of its arguments:

```
c(apple = 5030, banana = 4011, pear = 4421)
#>   apple  banana    pear
#>   5030    4011    4421
```

Notice now that the line has no [1] prefix, and that the names of the elements are listed above their values. R does an okay job of making sure that these stay aligned, even if their values are longer than their names:

```
c(a = "antediluvian", b = "boisterously", c =
"connoisseurs")
#>           a           b           c
#> "antediluvian" "boisterously" "connoisseurs"
```

The help file for `c()` (`?c`) shows that the definition of this function is `c(...)`, which means it captures any named or unnamed arguments. This function, therefore, converts these argument names into element names.

There are limitations to what the names can be, and these are the same as the limitations on variable names. Names must start with a letter, not a number. They can start with a dot (.) as long as it's not immediately followed by a number. The rest of the name can consist of letters (upper- and lowercase) and numbers, but not punctuation (except . or \_) or other symbols. Here the dot is slightly more appropriate (as you'll see, even common in some cases). The following are all valid names for elements:

```
c(high.value = 100, .score = 7, round2 = 26)
#> high.value      .score      round2
#>     100          7          26
```

But the following will fail:

```
c(.2b = 6)
#> Error: unexpected symbol in "c(.2b"
```

Again, you can bypass these restrictions by using quotes or backticks, but it means you need to use these to access the elements:

```
c(".2b" = 6)
#> .2b
#>   6

c("one fish" = "red", "two fish" = "blue")
#> one fish two fish
#>     "red"     "blue"
```

If you fail to name some elements (by not providing an argument name), then their name will be the empty string, which you won't see when you print the vector:

```
c(apple = 5030, 4011, pear = 4421)
#> apple      pear
#> 5030 4011 4421
```

Calling `str()` on a vector with names produces slightly different-looking output than one without names:

```
str(
  c(
    x = 7,
    y = 8,
    z = 9
  )
)
#> Named num [1:3] 7 8 9
#> - attr(*, "names")= chr [1:3] "x" "y" "z"
```

Now the output is prefixed with the word `Named`, which tells you that this vector has named elements. The rest of the first line matches what you would receive from the unnamed vector creation:

```
str(
```

```
c(7, 8, 9)
)
#> num [1:3] 7 8 9
```

If you fail to assign a name, `str()` shows that the empty string is being used:

```
str(
  c(apple = 5030, 4011, pear = 4421)
)
#> Named num [1:3] 5030 4011 4421
#> - attr(*, "names")= chr [1:3] "apple" "" "pear"
```

The usefulness of this feature is limited to values having a name you want to retain. Storing more details along with values requires a more complex structure, which you'll see shortly.

## 5.2 Sequences

It's likely you'll want to create a regularly increasing/decreasing vector of values at some point — a *sequence* — perhaps as an index of distinguishable objects, or identifiers. The values 1 through 5 can get tedious to enter manually, especially if you need a long sequence:

```
c(1L, 2L, 3L, 4L, 5L)
#> [1] 1 2 3 4 5
```

There's an easier way. The `seq()` (sequence) function takes start and end value inputs and generates a regular sequence between the two:

```
seq(from = 1, to = 5)
#> [1] 1 2 3 4 5
```

This has some clever default arguments, too, namely that the sequence starts or ends at 1, so you can neglect one of these and still generate a sequence:

```
seq(to = 5)
#> [1] 1 2 3 4 5
```

Leaving off all of the named arguments tends to work the way

you might expect (in many cases), but does so through a lot of guessing. If you try to generate a sequence from the input value 5 (because `from` is the first argument) to the default start value of 1, you get

```
seq(5)
#> [1] 1 2 3 4 5
```

which looks suspiciously like `seq(from = 1, to = 5)`. In this case, the function guesses that you want an increasing sequence and returns a sequence starting at 1 and ending at the input value.

## The importance of explicit arguments

Explicitly naming the arguments `from`, `to`, and `by` can save you a lot of time trying to figure out why a sequence doesn't look the way you expect. The defaults and internal code do a good job of guessing at what you're trying to create, but it's very easy to get caught up following a pattern. In the previous example, you created a sequence with `seq(5)` that assumes the default of `to = 1`. Because `from` is the first argument, and you didn't name one explicitly, this call assumes that you mean `from = 5`.

The fact that the sequence is increasing despite a request for a sequence from 5 to 1 is covered in the help page `?seq` with various examples of usage.

If you tried to modify the example and add a `by = 0.5` argument to generate the sequence in steps of 0.5, you would encounter an error:

```
seq(5, by = 0.5)
#> Error: wrong sign in 'by' argument
```

This is your fault, not R's. The code for `seq` covers various ways to call the function and does different things

depending on which arguments are present. In this case, `from` is given the value 5, `to` is given the default value 1, and you have therefore attempted to generate a sequence from 5 to 1 in steps of +0.5.

Requesting that the steps go in the right direction works fine:

```
seq(5, by = -0.5)
#> [1] 5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

In particular, if you are using negative values, you may want to be more explicit with your argument names:

```
seq(-3)
#> [1] 1 0 -1 -2 -3
```

The *type* of sequence generated (`numeric` or `integer`) depends on the arguments, but with a bit of guesswork regarding what you might be hoping to achieve. When arguments are missing, an `integer` sequence is returned by default, even though the input is `numeric`:

```
str(seq(5.0))
#> int [1:5] 1 2 3 4 5
```

If you change the step size using the argument `by` (even if it seems to be the same as the default), you can create a `numeric` sequence:

```
str(seq(5, by = -1)) ①
#> num [1:5] 5 4 3 2 1
```

①

**Note the numeric by value.**

Sequences don't have to be only whole numbers, though. They don't need to start or end at integers either. You can create a regular sequence that increases in steps of 0.2 with

```
seq(from = 1.2, to = 2.0, by = 0.2)
```

```
#> [1] 1.2 1.4 1.6 1.8 2.0
```

The `from` and `to` values also don't need to be equally spaced:

```
seq(from = pi, to = 6)
#> [1] 3.141593 4.141593 5.141593
```

Again, the `seq()` function will make some guesses and assumptions about what it is you want. Notice that the largest value in this sequence isn't equal to the `to` argument, but another step larger would be too large.

When sequences increase or decrease by 1, the shortcut function `:` makes writing these sequences easier. This is one of the special functions (such as `+`) that can be written between two values:

```
1 : 5
#> [1] 1 2 3 4 5
```

The spaces on either side of `:` are a matter of style, and most people will neglect them.

You may already be familiar with this notation from spreadsheet ranges. For example, `A1:A8` selects cells 1 through 8 in the first column, as shown in figure 5.3.

A1:A8	
	A
1	21
2	22
3	23
4	24
5	25
6	26
7	27
8	28

**Figure 5.3 A sequence of selected cells in a spreadsheet**

You've also seen this in the output of `str()` on a vector:

```
str(c(21, 22, 23, 24, 25))
#> num [1:5] 21 22 23 24 25
```

The `1:5` indicates the range of indices of the elements.

You can create reverse sequences just as easily

```
5 : 1
#> [1] 5 4 3 2 1
```

and non-integer sequences (which step by 1) too:

```
2.1 : 6.4
#> [1] 2.1 3.1 4.1 5.1 6.1
```

```
pi : 6
#> [1] 3.141593 4.141593 5.141593
```

```
3 : -3
#> [1] 3 2 1 0 -1 -2 -3
```

The use of `:` as a counter deserves special note. Often you'll want to create a counter that *iterates* over values. You might have three names to use in performing a calculation, and you could store them in a vector:

```
patientNames <- c("Thomas", "Richard", "Henry")
```

You might access these using the indices `1` to `3`. These indices form a regular sequence, so you may use `1:3`. If you aren't entirely sure how many values you might need, you can generalize it so that the last value is equal to however long the vector is, using the `length()` function.

```
1:length(patientNames)
#> [1] 1 2 3
```

This is all well and good as long as you have some values in the vector. But if an update to the vector removes them all, you can get into real trouble:

```
patientNames <- c() ①
length(patientNames) ②
#> [1] 0
```

```
1:length(patientNames) ③
#> [1] 1 0
```

(1)

**No entries**

(2)

**The length of the empty vector is 0.**

(3)

**The sequence 1:0 produces the values 1 and 0.**

You likely wanted the sequence to be empty, but because : accepts backward sequences, this produces a sequence two values long. Trying to do something with the zeroth and first values of an empty vector will probably lead to some unexpected results.

In cases like this, it is almost certainly better to use a function that behaves as you expect. There are two additional functions, seq\_len() and seq\_along(), that should be used instead of : when counting things. Each of these takes a single argument — either the desired length of the sequence or something to count — and creates a regular sequence starting at 1:

```
patientNames <- c("Thomas", "Richard", "Henry")
seq_len(length(patientNames))
#> [1] 1 2 3
```

```
seq_along(patientNames)
#> [1] 1 2 3
```

The big advantage to these is that if the thing whose elements you're counting becomes empty, the sequence generated from that is also empty:

```
seq_len(0)
#> integer(0)
```

```
seq_along(c())
#> integer(0)
```

## Sequence precedence

The sequence shortcut : has a place in the order of precedence, but it happens to sit between two other commonly used operators, one above, one below.

It's common to need to generate a sequence but then not count the last element. Generating the sequence can be as simple as `1:3`. If the upper limit was stored as a variable, that might be as follows:

```
lim <- 3
1:lim
#> [1] 1 2 3
```

If you later needed to reuse the sequence, but not include the last value, you might try

```
1:lim - 1
#> [1] 0 1 2
```

But the entire sequence has been shifted (it now starts at 0 and is still 3 elements long). The order of these operations is to perform the sequence calculation *first*, and then the subtraction (: is higher in the precedence list), so this is really equivalent to the following:

```
(1:lim) - 1
#> [1] 0 1 2
```

To change the `to` value of this sequence, add parentheses around just that component:

```
1:(lim - 1)
#> [1] 1 2
```

A different ordering happens, though, to raise the limit to a power:

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

Because exponentiation ( $\wedge$ ) is *higher* than : in the precedence list, this is equivalent to the following:

```
1:(lim ^ 2)
#> [1] 1 2 3 4 5 6 7 8 9
```

Again, strict use of parentheses will help avoid any issues over which operator takes precedence.

## 5.2.1 VECTOR FUNCTIONS

Now that you have some vectors, you'll probably want to do something with them. You can inspect vectors and determine their length with the `length()` function:

```
length(c(7, 8, 9))
#> [1] 3
```

```
length(c("dog", "cat"))
#> [1] 2
```

```
length(9:17)
#> [1] 9
```

You can calculate the sum of the elements just the same as if you passed them in one at a time:

```
sum(c(10, 20, 30))
#> [1] 60
```

```
sum(10, 20, 30)
#> [1] 60
```

You can now calculate the mean (average) of several values (recall from the brief comparison between `sum()` and `mean()` in section 4.1.6):

```
mean(c(1, 2, 3, 4))
#> [1] 2.5
```

You can also determine the minimum and maximum values in a vector:

```
min(c(91, 23, 59, 44))
#> [1] 23
```

```
max(c(91, 23, 59, 44))
#> [1] 91
```

Many functions will take multi-element vectors as input. Check the help file for any function you intend to try this with, though, because it's not guaranteed.

## 5.2.2 VECTOR MATH OPERATIONS

Be careful about mixing math operations with vectors, especially sequences, because the lack of parentheses around the `:` operation makes the meaning somewhat unclear. Suppose you wanted to perform an operation that required a sequence 1 longer than `patientNames`. You might try this:

```
patientNames <- c("Thomas", "Richard", "Henry")
1:length(patientNames) + 1
#> [1] 2 3 4
```

But that no longer starts at 1. The `:` operator has a higher *precedence* (recall section 3.2 on precedence) than `+`, so that operation is performed first (generating the sequence). Recall that every group of values is a vector, even if it has a length of 1, so the `+` function definitely knows how to add two vectors when both lengths are 1. When one or both of the lengths isn't 1, though, some more tricks come into play.

The preceding example features a sequence of length three (`1:3`, three being the length of `patientNames`) and another of length one (the value 1). The `+` function needs to return a sequence the same length as the longest of these two, so it *recycles* (repeats) the shorter value enough times to create another sequence of length three. You therefore really have the following:

```
c(1, 2, 3) + c(1, 1, 1)
#> [1] 2 3 4
```

When the lengths of the two vectors match, their elements can be

added by matching up their positions:

```
c(1, 2, 3)
# + + +
c(1, 1, 1)
# = = =
c(2, 3, 4)
```

The spreadsheet way of thinking about this operation would be to imagine adding the first elements and then “filling down” the calculation to the other elements. Spreadsheets “vectorize” the operation between the cells, as shown in [figure 5.4](#).

	A	B	C
1	1	1	=A1+B1
2	2	1	=A2+B2
3	3	1	=A3+B3

**Figure 5.4 Vectorized formulas in a spreadsheet performing addition between cells**

When the lengths don’t line up (one is not a multiple of the other), R complains (rightfully so) because it isn’t able to perform the operation cleanly:

```
c(1, 2, 3) + c(2, 3)
# Warning message:
# In c(1, 2, 3) + c(2, 3): longer object length is not a
multiple of
#> shorter object length
#> [1] 3 5 5
```

Here the shorter vector (`c(2, 3)`) has undergone *some* recycling, but not a complete repetition, so the operation looks like the following:

```
c(1, 2, 3)
# + + +
c(2, 3, 2)
# = = =
c(3, 5, 5)
```

If you’re working with longer vectors, the shorter vector recycles as many times as necessary to reach the length of the larger vector

```
c(1, 2, 3, 4, 5, 6) + c(4, 3)
#> [1] 5 5 7 7 9 9
```

which is really

```
c(1, 2, 3, 4, 5, 6)
# + + + + +
c(4, 3, 4, 3, 4, 3)
# = = = = =
c(5, 5, 7, 7, 9, 9)
```

All of this works perfectly well if you have stored these vectors in variables too:

```
x <- c(1, 2, 3)
y <- c(2, 3, 2)
x + y
#> [1] 3 5 5
```

**NOTE** Don't underestimate this last line. You've just performed a *vectorized operation* whereby the elements of two vectors were added pairwise without your having to explicitly tell the computer to add the first pair, add the second pair, and add the third pair — which in some other languages, you *do* need to do.

Eventually this will become second nature, but for now if you try to mentally unpack these whenever you write them, you'll save yourself a lot of worry. The fact that you can perform vectorized operations along with automatic recycling means you are able to turn some simple expressions into less-than-simple results:

```
x <- 1:10      ①
y <- c(1, 0)   ②
x * y
#> [1] 1 0 3 0 5 0 7 0 9 0
```

①

**length(x) is 10.**

②

**length(y) is 2, so it will be recycled to match the longer length.**

The preceding is equivalent to multiplying the (recycled) vector `1, 0, 1, 0, ..., 1, 0` by the equal-sized sequence `1, 2, 3, ..., 10`, so every other value becomes zero.

Vector recycling becomes even more powerful when one of the objects involved in the operation is of length 1. For example, to determine which of the following values were greater than 5, you could test this without recycling:

```
c(6, 5, 4, 5, 7, 4) > c(5, 5, 5, 5, 5, 5)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Or you could do this (by allowing the second vector to recycle):

```
c(6, 5, 4, 5, 7, 4) > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

If the vector is stored as a variable, this becomes simple and compact:

```
test_nums <- c(6, 5, 4, 5, 7, 4)
test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

You'll make frequent use of this feature when we get to the loop construct. Too often, people build a loop to perform some operation element by element when automatic vectorization will do this for you.

## 5.3 Matrices

Vectors are very handy for creating a single collection of grouped items, but sometimes you need more dimensions (for example, locating points on a grid — you need both horizontal and vertical identifiers). You could store the *x* and *y* coordinates of a set of points in individual vectors:

```
x <- c(0.26, 0.57, 0.84, 0.72)
y <- c(0.01, 0.58, 0.82, 0.96)
```

But this is prone to losing track of which values correspond to which. Instead, you can create a *matrix* of values — a two-dimensional grid of rows and columns. The function `matrix()` takes a data vector as its first argument, along with a specification of the number of rows and columns in which to store this data. You can store these `x` and `y` values by first combining them into a larger vector with `c()` and then specifying that you want to store these as a matrix with four rows (one for each pair of coordinates) and two columns (one for each of `x` and `y`):

```
xy <- matrix(data = c(x, y), nrow = 4, ncol = 2)
xy
#>      [,1] [,2]
#> [1,] 0.26 0.01
#> [2,] 0.57 0.58
#> [3,] 0.84 0.82
#> [4,] 0.72 0.96
```

The `matrix()` function will populate this matrix one column at a time by default. If the data is arranged such that it needs to be entered one row at a time, there is an optional argument `byrow`. Note that because the input is a vector (created using `c()` here), the elements must *all* be of the same type. This is a restriction on the `matrix` object.

R will do the required division to spread out data to the number of rows or columns you ask for:

```
matrix(1:8, nrow = 2)
#>      [,1] [,2] [,3] [,4]
#> [1,]     1     3     5     7
#> [2,]     2     4     6     8

matrix(1:8, nrow = 4)
#>      [,1] [,2]
#> [1,]     1     5
#> [2,]     2     6
#> [3,]     3     7
#> [4,]     4     8
```

But the division must work out:

```
matrix(1:7, nrow = 4)
# Warning message:
# In matrix(1:7, nrow = 4): data length [7] is not a sub-
multiple or
#> multiple of the number of rows [4]
#>      [,1] [,2]
#> [1,]    1    5
#> [2,]    2    6
#> [3,]    3    7
#> [4,]    4    1
```

In this case, you've once again encountered *recycling* — this time of the entire data vector, which has been expanded to fit the requested shape (a rectangle with four rows).

This structure should begin to look familiar if you're accustomed to working with raw data in a spreadsheet as a table, as shown in [figure 5.5](#).

	A	B	C	D
1	1	3	5	7
2	2	4	6	8

**Figure 5.5 A table of values in a spreadsheet, similar to a matrix**

The `str()` function applied to a matrix tells you about each of the dimensions (note that still only a single type is reported, because all elements must be of this type):

```
str(
  matrix(1:8, nrow = 2)
)
#> int [1:2, 1:4] 1 2 3 4 5 6 7 8
```

These dimensions are also recorded in an attribute of the matrix with the label `dim`

```
attributes(matrix(1:8, nrow = 2))
#> $dim
#> [1] 2 4
```

where the number of rows and columns are stored in a vector (respectively).

### 5.3.1 NAMING DIMENSIONS

The `xy` matrix has the values of `x` as the first column and the values of `y` as the second column. The rows and columns have descriptions printed alongside them: rows are indexed as `[row, ]`, and columns are indexed as `[, col]`. R counts each of these, starting from 1, using numbers in both directions (horizontal and vertical). This is an important distinction if you are coming from a spreadsheet program that tends to label the rows with integers and the columns with letters.

In a spreadsheet, you may refer to a cell as A5, meaning the cell in the fifth row of the first column. R uses numbers in both positions, so that element would correspond to the `[5, 1]` (row 5, column 1) position in the matrix.

The `str()` function displays the *type* of the matrix (the common type of the elements, since they must all be the same) and the length of the rows and columns:

```
str(xy)
#> num [1:4, 1:2] 0.26 0.57 0.84 0.72 0.01 0.58 0.82 0.96
```

Because the `data` argument is a vector, and all elements of a vector need to be of the same type, this coercion occurs before a matrix is even created:

```
matrix(c("a", 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
#> [,1] [,2]
#> [1,] "a"  "4"
#> [2,] "2"  "5"
#> [3,] "3"  "6"
```

Hence, a single type is printed by the `str()` function.

To make this easier to read, you can also specify the names of the columns using the `colnames()` function. Because you didn't tell the `matrix()` function what to use for column names when defining `xy` earlier, it doesn't have any:

```
colnames(xy)
#> NULL
```

This function behaves differently, though, when you follow it with an assignment operator (`<-`) in that it can assign a vector to

the names of the columns. Every operation in R is a function, and this is no exception; there is another function named `colnames<-` that performs this task, but R is clever enough to know that this is what you want to do:

```
colnames(xy) <- c("x", "y")
xy
#>      x     y
#> [1,] 0.26 0.01
#> [2,] 0.57 0.58
#> [3,] 0.84 0.82
#> [4,] 0.72 0.96
```

If you try to assign names with a vector of the wrong length, R *won't* recycle the names:

```
colnames(xy) <- c("z")
#> Error: length of 'dimnames' [2] not equal to array
extent
```

Correctly setting the `colnames` of a matrix also sets another attribute: `dimnames`. This has a more complex structure, but it essentially stores the names of the dimensions.

There is also a `rownames()` function (and corresponding `rownames<-`). There are limited cases where using this will be a good idea, but it's certainly possible. It behaves in the same way as `colnames()` and sets the names of the rows:

```
rownames(xy) <- c("r1", "r2", "r3", "r4")
xy
#>      x     y
#> r1 0.26 0.01
#> r2 0.57 0.58
#> r3 0.84 0.82
#> r4 0.72 0.96
```

## 5.4 Lists

When you need to store things that aren't necessarily of the same length or type, turn to *lists*. These are more complex structures, but with that complexity comes flexibility and usefulness.

Technically, lists are vectors, but it will quickly become clear that there's a lot more to them. You create lists with the unsurprisingly named `list()` function. You can create a very simple list like this:

```
list(7, 8, 9)
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 8
#>
#> [[3]]
#> [1] 9
```

The output looks very different from that of a vector at first glance, but there are some similarities. You still have the `[1]` output, though now that appears next to *each* value, rather than just in front of the first. In addition, you now have a double-bracketed counter `[[1]]`, `[[2]]`, and `[[3]]`.

The argument to the `list()` function is . . . , which, as you saw in section 4.1.6, takes arguments of any (or no) name and passes them on to some internals. In the same way as with the `sum()` function in that section, each *object* you provide as an argument is taken individually, so in this case the `list()` function has created a list with three elements: 7, 8, and 9. Each of these, as you saw at the start of this chapter, is its own vector (of length 1), hence the `[1]` at the start of each listing.

Section 5.1 said that in a vector, the elements need to all be of the same type, or else they will be coerced to being so. Lists don't have this restriction, so you can create mixed-type lists without risking coercion:

```
list(2L, "a", 5.2)
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
```

```
#> [1] 5.2
```

The true benefit of lists comes when you store longer vectors. These individually follow the rules for their type (for vectors, elements must be of the same type or are coerced), but different list elements can be entirely different in both length and type:

```
list(
  c(1.2, 4.8),
  c(3L, 5L, 7L, 9L),
  c("cat", "dog", "mouse")
)
#> [[1]]
#> [1] 1.2 4.8
#>
#> [[2]]
#> [1] 3 5 7 9
#>
#> [[3]]
#> [1] "cat"    "dog"    "mouse"
```

You can mix in shorthand notation for sequences, or anything you like (even more lists!). You can also name the elements of the list, which can be handy for identifying what the elements represent:

```
list(
  nameMatrix = matrix(c("a", 2, 3, 4, 5, 6), nrow = 3, ncol = 2),
  commonValue = NA,
  index = 1:8
)
#> $nameMatrix
#>      [,1] [,2]
#> [1,] "a"   "4"
#> [2,] "2"   "5"
#> [3,] "3"   "6"
#>
#> $commonValue
#> [1] NA
#>
#> $index
#> [1] 1 2 3 4 5 6 7 8
```

The generic [[1]], [[2]], [[3]] labels are replaced with the

element name, prefixed with a \$. This “dollar-name” format will be a useful way to access these elements later.

The `NULL` element cannot appear in a vector, but it’s a different story for a list:

```
list(
  x = c(4, 5, 6),
  y = NULL,
  z = c(7, 8, 9)
)
#> $x
#> [1] 4 5 6
#>
#> $y
#> NULL
#>
#> $z
#> [1] 7 8 9
```

For a vector to hold a `NULL` element, it would need to coerce “nothing” into something (or vice versa), which can’t be done. A list, however, has no issue with one or more of its elements being “nothing” (`NULL`).

The `str()` function applied to a list reports each of the vector types and provides any names that have been defined:

```
str(
  list(
    a = c(1.2, 2.3, 3.4),
    b = 2:5,
    c = c("alpha", "omega")
  )
)
#> List of 3
#> $ a: num [1:3] 1.2 2.3 3.4
#> $ b: int [1:4] 2 3 4 5
#> $ c: chr [1:2] "alpha" "omega"
```

**NOTE** There isn’t a spreadsheet equivalent of R’s lists because spreadsheets don’t specifically contain groups of data — typically, just lots of individual cells. Because a worksheet is a grid of cells, and those cells can individually do what you need,

you can spread out data of any and all types wherever you like. This lack of structure tends to enable people to place “tables” of data all over sheets, overlapped by graphs, calculations, and notes.

## 5.5 *data.frames*

Keeping track of the positions along each list element’s vector for multifaceted data would soon become insurmountable. As was the case when you expanded from a vector to a matrix, you can increase the dimensions of lists and create something more tabular. The `data.frame` is the most common and useful type of object in R, but to truly understand it, you’ve needed to see all the predecessors first.

If you have a list of vectors, each having its own (consistent) type but all of the same length (or recyclable to the same length), then you can store that information in a table layout (rows and columns) using `data.frame()`:

```
data.frame(
  list(
    x = c(1, 2, 3, 4),
    y = c("a", "b"),
    z = c(2.1, 9.3, 7.6, 1.1)
  )
)
#>   x y   z
#> 1 1 a 2.1
#> 2 2 b 9.3
#> 3 3 a 7.6
#> 4 4 b 1.1
```

①

**Creates a list with three elements (vectors) named "x", "y", and "z"**

②

**The y vector is only of length 2, so it will be recycled to match the length 4 vectors.**

---

R is clever enough to not actually need that `list()` call; you can send the vectors as arguments to `data.frame()`:

```
data.frame(
  x = c(1, 2, 3, 4),
  y = c("a", "b"),
  z = c(2.1, 9.3, 7.6, 1.1)
)
#>   x y   z
#> 1 1 a 2.1
#> 2 2 b 9.3
#> 3 3 a 7.6
#> 4 4 b 1.1
```

This structure should be familiar if you've worked with spreadsheet tables; column names and data are all here, but now instead of referring to rows and columns with labels like A1, you have the column numbers/names and row numbers (potentially names, too). A spreadsheet view of this might look like figure 5.6.

	A	B	C
1	x	y	z
2		1 a	2.1
3		2 b	9.3
4		3 a	7.6
5		4 b	1.1

**Figure 5.6 A spreadsheet table of cells**

Technically, the names for the vectors aren't required either, but it's dangerous to not supply them; if names aren't supplied for the vectors, names will be automatically created based on the way the values were supplied, with some care taken about spaces and other disallowed characters in the names, and made to be unique. Using the backtick/quote syntax for bypassing the naming rules leads to less-than-pretty column names that aren't particularly useful down the road:

```
data.frame(c(1, 2), "'oh no!'")
#>   c.1..2. X..oh.no...
#> 1      1    'oh no!'
#> 2      2    'oh no!'
```

Once again, the special variable names can be applied here, as

long as they are suitably quoted. It's common for spreadsheets to have columns named with spaces, numbers, or punctuation, which R doesn't allow. One option is to use the quoting approach:

```
data.frame(
  x = 21,
  "y variable" = 22,
  ".2b" = 23
)
#>     x y.variable X.2b
#> 1 21          22   23
```

R allows you to create the `data.frame`, but it draws the line at keeping those names in the improper format. Spaces and other unallowed punctuation are replaced with dots, whereas any names starting with a number (or dot-number) are renamed to start with an x.

If you *really* want to turn off this conversion (though it may very well mean a `data.frame` is not compatible with some other functions), you can do so with the argument `check.names = FALSE`:

```
data.frame(
  x = 21,
  "y variable" = 22,
  ".2b" = 23,
  check.names = FALSE
)
#>     x y variable .2b
#> 1 21          22   23
```

Because the columns are defined via a `list`, which is a group of vectors, each column can have a different type (though each element in a column must be of a consistent type because it is a vector). This is one of the ways in which a `data.frame` differs from a `matrix`, where *all* the elements in all the rows and columns must be of the same type.

The requirement that the vectors all be the same length means there will guaranteeably be some number of rows in the `data.frame` (even if that number is zero). If you supply an

incompatible set of vectors (for which the shorter can't be recycled to the longer length), then R complains and gives up:

```
data.frame(x = c(3, 4), y = c("q", "r", "s"))
#> Error: arguments imply differing number of rows: 2, 3
```

R is perfectly okay with creating a `data.frame` with no rows:

```
data.frame()
#> data frame with 0 columns and 0 rows
```

You could try to add some named (but empty) columns:

```
data.frame(x = c(), y = c(), z = c())
#> data frame with 0 columns and 0 rows
```

But the empty vector `c()` is the same as `NULL`, which can't be compared to anything or contribute to a vector, so it can't be used as a `data.frame` element. Instead, you can create 0-element named vectors<sup>3</sup> and use these in the call:

---

<sup>3</sup> `integer()`, `character()`, and `numeric()` each have an argument `length` with default value 0.

```
data.frame(
  x = integer(),
  y = character(),
  z = numeric()
)
#> [1] x y z
#> <0 rows> (or 0-length row.names)
```

“Empty” `data.frames` like this aren’t entirely useless, as you’ll see later. Being able to store the name of the columns even before any data is available can make working with `data.frames` much easier.

You provide vectors to `data.frame()` to be used as columns, and these columns will have the type of each vector with one very important distinction: if you use a character vector, then `data.frame()` will automatically convert this to a factor. If you create a `data.frame` using a character vector and a numeric

vector, the character column is converted to a factor:

```
char_and_num_df <- data.frame(
  x = c("apple", "banana", "carrot"),
  y = c(5, 6, 7)
)
str(char_and_num_df)
#> 'data.frame':      3 obs. of  2 variables:
#>   $ x: Factor w/ 3 levels "apple","banana",...: 1 2 3
#>   $ y: num  5 6 7
```

This causes a lot of confusion, but it's the default behavior. To override this, set `stringsAsFactors = FALSE` when creating the `data.frame`:

```
char_and_num_df <- data.frame(
  x = c("apple", "banana", "carrot"),
  y = c(5, 6, 7),
  stringsAsFactors = FALSE    ①
)
str(char_and_num_df)
#> 'data.frame':      3 obs. of  2 variables:
#>   $ x: chr  "apple" "banana" "carrot"
#>   $ y: num  5 6 7
```

①

**Don't coerce character vectors to factor columns. The default for this argument is TRUE.**

You can set this to be the default using the following option in a session before creating a `data.frame`:

```
options(stringsAsFactors = FALSE)
```

Those column names are stored in an attribute (introduced in section 5.1.3), but you may have noticed that R doesn't tell you this in the same way it did when you printed a vector with an attribute. That's because R has some additional code it refers to when you ask it to print a `data.frame` to the Console. In order to see these column names, use the `attributes()` function:

```
attributes(mtcars)
#> $names
```

```

#> [1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec"
"vs"   "am"   "gear"
#> [11] "carb"
#>
#> $row.names
#> [1] "Mazda RX4"          "Mazda RX4 Wag"      "Datsun
710"
#> [4] "Hornet 4 Drive"     "Hornet Sportabout"
"Valiant"
#> [7] "Duster 360"        "Merc 240D"       "Merc
230"
#> [10] "Merc 280"          "Merc 280C"       "Merc
450SE"
#> [13] "Merc 450SL"        "Merc 450SLC"
"Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial" "Fiat
128"
#> [19] "Honda Civic"       "Toyota Corolla"   "Toyota
Corona"
#> [22] "Dodge Challenger"  "AMC Javelin"    "Camaro
Z28"
#> [25] "Pontiac Firebird"  "Fiat X1-9"
"Porsche 914-2"
#> [28] "Lotus Europa"      "Ford Pantera L"
"Ferrari Dino"
#> [31] "Maserati Bora"     "Volvo 142E"
#>
#> $class
#> [1] "data.frame"

```

You see the attributes `names`, `row.names`, and `class`. First are the column names, and next are the names of the rows. Row names are a peculiar feature, and I don't recommend getting too attached to them. If you create a `data.frame` and print it (which R does by default), you see the rows labelled by their indices 1, 2, and 3:

```

data.frame(
  col1 = c("x", "y", "z"),
  col2 = c("q", "r", "s")
)
#> col1 col2
#> 1     x     q
#> 2     y     r
#> 3     z     s

```

If you print the `mtcars` data set, however, it too is arranged as a series of columns with names, but the first column appears to have no name and the indices are missing. That's because this particular data set has *labelled rows* (`row.names`). You can see these by asking for them:

```
rownames(mtcars) ①
#> [1] "Mazda RX4"          "Mazda RX4 Wag"      "Datsun
710"
#> [4] "Hornet 4 Drive"    "Hornet Sportabout"
"Valiant"
#> [7] "Duster 360"        "Merc 240D"        "Merc
230"
#> [10] "Merc 280"          "Merc 280C"        "Merc
450SE"
#> [13] "Merc 450SL"        "Merc 450SLC"
"Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial"  "Fiat
128"
#> [19] "Honda Civic"       "Toyota Corolla"    "Toyota
Corona"
#> [22] "Dodge Challenger"  "AMC Javelin"      "Camaro
Z28"
#> [25] "Pontiac Firebird"  "Fiat X1-9"
"Porsche 914-2"
#> [28] "Lotus Europa"      "Ford Pantera L"
"Ferrari Dino"
#> [31] "Maserati Bora"     "Volvo 142E"
```

①

`row.names()` (with the period) also works. These aren't precisely the same functions, though. Don't ask why.

That returns a vector of the row names. This is an odd design, however — the names of the rows are not *part* of the data but are merely attributes of the data. I don't recommend following this pattern (though it's good to recognize it). Instead I find it much more suitable to have these values as a character column in the `data.frame`.

The last attribute of the `mtcars` data set in the output of `attributes()` was `class`, and this is a much more important attribute than it may appear at first glance.

## 5.6 Classes

You've already seen that R behaves differently when asked to print a vector, a matrix, or a `data.frame`:

```
print(c(4, 5, 6))
#> [1] 4 5 6

print(matrix(c(4, 5, 6)))
#>      [,1]
#> [1,]     4
#> [2,]     5
#> [3,]     6

print(data.frame(a = c(4, 5, 6)))
#>   a
#> 1 4
#> 2 5
#> 3 6
```

But how does R know to do this in each case? You called the same `print()` function each time, right?

Objects may have an attribute `class` that labels the structure that object represents. A `matrix` has the `class` "matrix", and a `data.frame` has the `class` (can you guess?) "data.frame".

You print these objects to the Console by calling the `print()` function with the object as the argument (as earlier). When R sees these calls, it first checks whether the `class` attribute has been set for the object. If it has (say, to "someClass"), R looks through the functions it knows about and checks whether any of their names match the pattern `print.someClass`. These extensions of a function that depend on the `class` are known as *methods*.

**DEFINITION** A *method* is a function that only applies to a particular class: for example, the `print()` method for the class "matrix". Different versions of the same *generic* function (in this case, `print()`) can be defined for different classes, and R knows to use these if the `class` attribute of the object passed into

the first argument is assigned.

If R finds a `print()` method matching the class of the object, it uses that to print the object. If it doesn't find one, it calls `print.default()` — the default print method.

You can call any of the methods explicitly if you like. For example:

```
d <- data.frame(x = 1:3, y = c(3.6, 2.7, 0.4))
print.data.frame(d)
#>   x     y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4

print.default(d)
#> $x
#> [1] 1 2 3
#>
#> $y
#> [1] 3.6 2.7 0.4
#>
#> attr(,"class")
#> [1] "data.frame"
```

**CAUTION** It may be clearer now why I suggest *not* using a period in the name of a function — does the `print.data.frame` method call `print()` for an object with `class "data.frame"` or does it call the `print.data()` method for an object with `class "frame"`? It's the former, though you can certainly construct the other scenario and become confused.

This is a very simple system, and if it seems to be saying not much more than “This should behave slightly differently if it has this label on it,” it’s because that’s a fairly accurate representation. Nothing is really stopping you from changing how R sees the class of an object (for your own sanity, you don’t want to have to do this regularly) and thus how R thinks it

should print the object:

```
x <- c(4, 5, 6)      ①
class(x) <- "data.frame"  ②
print(x)      ③
#> NULL
#> <0 rows> (or 0-length row.names)
```

①

**Not a data.frame**

②

**But let's tell R that it is one by setting an attribute.**

③

**Because x really isn't a data.frame, the print.data.frame method can't handle this.**

You aren't limited to the classes that R already knows about — to create a more complex structure that behaves differently from a data.frame, add more attributes on top of existing ones. R will try each of these in order until it finds a method it can use, skipping over any that it doesn't recognize:

```
class(d) <- c("specialObject", "data.frame")
str(d)
#> Classes 'specialObject' and 'data.frame':    3 obs. of
2 variables:
#>   $ x: int  1 2 3
#>   $ y: num  3.6 2.7 0.4

print(d)
#>   x     y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4
```

To change this behavior, define the method yourself:

```
print.specialObject <- function(x, ...) {
  cat("## I'm more than just a data.frame!\n")
  print.data.frame(x)
```

```

}
print(d)
#> ## I'm more than just a data.frame!
#>   x   y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4

```

The rules surrounding how this can be done are beyond the scope of this book, but knowing that this occurs will be vitally important.

### 5.6.1 THE TIBBLE CLASS

The `data.frame` is a fundamental unit in the majority of R package functions. That said, there's certainly room for improvement with regard to a few of that structure's features.<sup>4</sup> One very useful extension in this space is the `tibble` package,<sup>4</sup> which provides a new structure `tibble`, similar to `data.frame` but with some carefully crafted improvements.

---

<sup>4</sup> Originally this was a part of Hadley Wickham's `dplyr` package as `tbl_df`, for table data frame, but pronounced "tibble," and was extracted into its own package by Kirill Müller.

First, install, load, and attach the package:

```
# install.packages("tibble")
library(tibble)
```

To quote from the help page `?tibble`, the advantages that `tibbles` have over `data.frames` are that a `tibble`

- Never coerces inputs — strings stay as strings! Compare to `stringsAsFactors = TRUE`.
- Never adds `row.names`. These shouldn't be used. Compare to the `mtcars` dataset.
- Never munges column names. Compare to `data.frame("my column" = c(1, 2, 3))`, which creates a column `my.column`.
- Only recycles length 1 inputs. Compare to `data.frame(x = c(1,`

`2, 3, 4), y = c("a", "b"))`, which recycles the `y` column.

- Evaluates its arguments lazily and in order. Copies of objects are only made when necessary, which saves computation time.
- Adds the `tbl_df` class to output. `tibbles` are identified by the `classestbl_df`, which means distinct methods can be applied.

In addition to those construction improvements, the package also provides an improved `print` method (`print.tbl_df()`) that itself has several improvements over the `print()` method for `data.frames`:

- Rows and columns that don't fit into the Console window aren't shown, but are noted as such. Compare to the `print.data.frame()` method, which wraps the output around after a certain number of rows, leaving you scrolling back and forth between blocks.
- The total number of rows and columns is printed before the data.
- The type of each column is printed below the column name.
- Numeric precision is highlighted using shading (where supported).
- Negative numbers are colored red (where supported).
- Missing entries are highlighted with yellow (where supported).

The `tibbles` printed in this book won't show these features, but if you're using a fairly recent version of RStudio, then the output of

```
tibble(
  chars = letters[1:5],
  nums = -2:2,
  missing = c("x", NA, "y", NA, "NA"),
  precise = c(1.00002, 1.0002, 100.002, 10.200, 0.002)
)
```

should look something like figure 5.7.

	<code>chars</code>	<code>nums</code>	<code>missing</code>	<code>precise</code>
	<code>&lt;chr&gt;</code>	<code>&lt;int&gt;</code>	<code>&lt;chr&gt;</code>	<code>&lt;dbl&gt;</code>
1	a	-2	x	1.00
2	b	-1	NA	1.00
3	c	0	y	100
4	d	1	NA	10.2
5	e	2	NA	0.00200

**Figure 5.7 Helpful printing of different data features in tibbles**

These features alone make tibbles a vastly superior object class, but there are further advantages in how selection behaves.

If all this isn't enough to convince you of the advantage tibble has over `data.frame`, there's one more mind-blowing feature that makes this package even more useful. You saw earlier that a `data.frame` is essentially a list of vectors, all of the same length. Those vectors can contain any basic type of value, be it numeric, character, Date, or other. `tibble` extends this concept to the idea of `list-columns`: rather than a vector of values, a `tibble` column consists of a list of objects. These can be anything, because they're in a list rather than a vector, so you can create a `tibble` with all sorts of data contained in a column, provided the length of each top-level list is the same.

You can create a `tibble` with a list-column just as easily as creating a `data.frame`:

```
tibble(
  x = 1:3,          ①
  y = list(letters, TRUE, mtcars)  ②
)
#> # A tibble: 3 × 2
#>       x     y
#>   <int> <list>
#> 1     1 <chr [26]>
#> 2     2 <lgl [1]>
#> 3     3 <data.frame [32 × 11]>
```

①

The first column contains a vector of numeric values.

②

The second column contains a character vector, a logical vector, and a data.frame, together stored in a list.

You can convert lists of lists to tibbles with the `as.tibble()` function:

```
my_tbl <- as.tibble(  
  list( ②  
    x = 1:3, ③  
    y = list( ④  
      letters, ⑤  
      TRUE, ⑥  
      mtcars ⑦  
    )  
  )
```

②

It contains a list of lists (a list of columns, some of which may themselves be lists).

③

The first column is a sequence from 1 to 3.

④

The second column is a list-column containing different types.

⑤

A character vector of length 26

⑥

A logical vector of length 1

⑦

The mtcars dataset (a data.frame)

The `print` method for `tbl_df` is automatically called when you

print the tibble:

```
my_tbl
#> # A tibble: 3 × 2
#>   x     y
#>   <int> <list>
#> 1     1 <chr [26]>
#> 2     2 <lgl [1]>
#> 3     3 <data.frame [32 × 11]>
```

The following additional information is available:

- The number of rows and columns (three rows by two columns)
- The type of each column (`int` and `list`)
- The type and length of each element of the `list`-column (for example, `<chr [26]>`)

Suppose you have an object with lots of information, such as this `data.frame` made up of 26 rows and 26 columns:

```
m <- as.data.frame(
  matrix(
    1:676, c(26, 26),
    dimnames = list(1:26, LETTERS)
  )
)
```

You can `print()` it (I'm only showing the first six rows and that's bad enough):

```
head(m)
#>   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
P   Q   R   S
#> 1  1  27  53  79 105 131 157 183 209 235 261 287 313 339 365
391 417 443 469
#> 2  2  28  54  80 106 132 158 184 210 236 262 288 314 340 366
392 418 444 470
#> 3  3  29  55  81 107 133 159 185 211 237 263 289 315 341 367
393 419 445 471
#> 4  4  30  56  82 108 134 160 186 212 238 264 290 316 342 368
394 420 446 472
#> 5  5  31  57  83 109 135 161 187 213 239 265 291 317 343 369
395 421 447 473
#> 6  6  32  58  84 110 136 162 188 214 240 266 292 318 344 370
396 422 448 474
```

```
#>      T   U   V   W   X   Y   Z
#> 1 495 521 547 573 599 625 651
#> 2 496 522 548 574 600 626 652
#> 3 497 523 549 575 601 627 653
#> 4 498 524 550 576 602 628 654
#> 5 499 525 551 577 603 629 655
#> 6 500 526 552 578 604 630 656
```

If you convert to a `tibble`, though, rather than filling up the Console, you are only shown the information that fits, with some notes on what was left out:

```
as.tibble(m)
#> # A tibble: 26 x 26
#>       A     B     C     D     E     F     G     H     I
#>       J     K     L
#> * <int> <int> <int> <int> <int> <int> <int> <int>
#> <int> <int> <int>
#> 1    1    27    53    79   105   131   157   183   209
#> 235 261 287
#> 2    2    28    54    80   106   132   158   184   210
#> 236 262 288
#> 3    3    29    55    81   107   133   159   185   211
#> 237 263 289
#> 4    4    30    56    82   108   134   160   186   212
#> 238 264 290
#> 5    5    31    57    83   109   135   161   187   213
#> 239 265 291
#> 6    6    32    58    84   110   136   162   188   214
#> 240 266 292
#> 7    7    33    59    85   111   137   163   189   215
#> 241 267 293
#> 8    8    34    60    86   112   138   164   190   216
#> 242 268 294
#> 9    9    35    61    87   113   139   165   191   217
#> 243 269 295
#> 10   10   36    62    88   114   140   166   192   218
#> 244 270 296
#> # ... with 16 more rows, and 14 more variables: M <int>,
#> N <int>, O <int>,
#> # P <int>, Q <int>, R <int>, S <int>, T <int>, U
#> <int>, V <int>,
#> # W <int>, X <int>, Y <int>, Z <int>
```

Here, 16 rows and 14 columns (and their names) aren't shown. Resizing the Console changes what's printed (the next time it's

printed) too.

In the list of advantages of tibbles, I mentioned that they never munge column names. Remember that in order to have `data.frame` not convert spaces and punctuation to dots, you need to specify `check.names = FALSE`. You have no such trouble when working with tibbles:

```
tibble(
  x = 21,
  `y variable` = 22,
  `.2b` = 23
)
#> # A tibble: 1 × 3
#>       x `y variable`  `.2b`
#>   <dbl>      <dbl> <dbl>
#> 1  21.0      22.0  23.0
```

You may not need to create a tibble from scratch (there are very neat ways to do so), but once you start working with `data.frames` more, you'll use functions that return this structure, so it's essential to know about them.

## 5.6.2 STRUCTURES AS FUNCTION ARGUMENTS

When you built and called functions in the last chapter, you only passed single values as inputs. Now that you know how to make more advanced structures, how do these work with functions?

You'll be happy to know that there's nothing new to learn — functions can be made to receive *any* structure as input (whether they're able to do something with the input is an entirely different matter). In fact, because there's no such thing in R as "just a value," you've been passing vectors (of length 1) to functions the entire time!

Most base R functions work well with vectors of length greater than 1 by default.<sup>5</sup> The `sqrt()` function calculates the square root of its input:

---

<sup>5</sup> They *usually* deal with vectors, because a length 1 value is still a vector.

---

```
sqrt(9)
#> [1] 3
```

It will gladly perform the same operation on a (longer) vector of inputs, returning a new vector of outputs:

```
sqrt(c(9, 16, 25))
#> [1] 3 4 5
```

The structural requirements of function arguments are specific to that function, but *should* be explained in sufficient detail in the help file for the function. If you’re writing your own function, then it’s up to you which input structures to support or require. Make a sensible choice based on what your function *needs*; perhaps you need an entire `data.frame`, a list, or just a vector. Try to have as input the minimal amount of data for the function to achieve its task.

To be rigorous about the input structure, consider adding some logic to test the input type with `class()` or `typeof()`. You’ll see how to do this when we cover conditional evaluation.

Finally, the *output* of a function (the argument to `return()`) can just as easily be any structure. A function can only return one object, but that need not stop you from building a list of individual pieces to return. For example, the returned object from `lm(1~1)` (a trivial linear regression — the details aren’t important here) is a list containing 11 parts:

```
str(lm(1~1))
#> List of 11
#> $ coefficients : Named num 1
#>   ... attr(*, "names")= chr "(Intercept)"
#> $ residuals     : Named num 0
#>   ... attr(*, "names")= chr "1"
#> $ effects       : Named num 1
#>   ... attr(*, "names")= chr "(Intercept)"
#> $ rank          : int 1
#> $ fitted.values: Named num 1
#>   ... attr(*, "names")= chr "1"
#> $ assign         : int 0
#> $ qr             :List of 5
```

```
#>   ..$ qr    : num [1, 1] 1
#>   ... - attr(*, "dimnames")=List of 2
#>   ...   ..$ : chr "1"
#>   ...   ..$ : chr "(Intercept)"
#>   ... - attr(*, "assign")= int 0
#>   ..$ qraux: num 1
#>   ..$ pivot: int 1
#>   ..$ tol  : num 1e-07
#>   ..$ rank : int 1
#>   ... - attr(*, "class")= chr "qr"
<truncated>
```

## 5.7 Try it yourself

Create your own vector of values. Can you mix different types (integers and characters)? Create another vector that has the values 10, 9, ..., 1. Did you use the `seq()` function? Can you use the `:` shortcut?

Try subtracting two equal-length vectors, and do the element-wise subtraction yourself to see if R arrives at the answer you expect. What if you multiply two vectors? What if they aren't the same length?

Create a list that represents three qualities about yourself (such as your name, address, and date of birth). Split some of the parts of these qualities out into separate elements within each list element. Are the types what you expect?

Create a `data.frame` with three rows and four columns, where each row represents a person, and the columns represent that person's name (character), age (integer), whether or not they like red wine (logical), and their height in meters (numeric). It might look something like this:

```
data.frame(
  name = c("Alice", "Bob", "Charlie"),
  age = c(21L, 43L, 19L),
  likes_red_wine = c(TRUE, FALSE, TRUE),
  height_m = c(1.65, 1.50, 1.75)
)
#>      name age likes_red_wine height_m
#> 1  Alice  21        TRUE     1.65
```

#> 2	Bob	43	FALSE	1.50
#> 3	Charlie	19	TRUE	1.75

Now create the same `data.frame` but acknowledge that you don't know whether or not Bob likes red wine by using `NA`. Assign this `data.frame` to a variable called `people`. Use `str()`, `print()`, `class()`, and anything else you know about to investigate this new object. Convert `people` to a `tibble` and print it. Do you notice anything? Did you remember to load the `tibble` package with `library(tibble)`?

## Terminology

- *Vector* — A simple structure where all elements have the same type.
- *Sequence* — A vector of regularly spaced elements.
- *Recycle* — When the lengths of two vectors differ, one is repeated (recycled) to the longer length.
- *Matrix* — Grid of elements, all of the same type, laid out in rows and columns.
- *List* — Similar to a vector of vectors, but each list element can be a different type and length.
- `data.frame` — A list of vectors where each vector is the same length but may be a different type, laid out in a grid of rows and columns.
- *Class* — Label of a particular structure that allows R to identify which functions (methods) to apply.
- `tibble` — An improved `data.frame` with additional functionality (such as list-columns).

## Summary

- You've been working with vectors all along; R has no concept of a "scalar" value.

- Vectors and matrices contain a single type of data.
- NA can be included in a vector, but not NULL.
- Names can be attached to elements of a vector.
- Integer-spaced sequences can be generated using :.
- Not every function expects a vector or expects individual values.
- When two vectors are involved in a comparison, the shorter may be recycled to the longer length.
- Lists can combine different types of data.
- A `data.frame` will have a consistent number of rows in each column.
- Different classes have different methods, such as `print.table()`.
- The `tibble` structure is an improvement over `data.frame`.
- All elements of vectors and matrices must be of a single type, coerced to be the most general of those available.
- A vector or matrix can't contain NULL, but it can contain the relevant flavor of NA.
- A list can contain NULL as a top-level element, but it still can't be part of a vector or matrix.
- The `seq()` function will assume you mean to start at 1 if you don't specify enough arguments.
- The shortcut function `:` will gladly count backward, whether you want to or not, as in `1:0`.
- Column names of a `data.frame` can't contain spaces or punctuation.
- A `tibble` is printed to the Console in a much more useful way than a `data.frame`.

# 6

## *Selecting data values*

---

### This chapter covers

- Manipulating strings
- Selecting groups of values from different structures
- Conditionally selecting values
- Replacing certain values
- Manipulating `data.frames` in a better way: `dplyr`

Now that you've learned how to group values together into larger structures, it's going to be useful to access smaller components of those from time to time, be it a column from a `data.frame`, a row from a matrix, or part of a string. Extracting data from larger structures is both one of the most common and most dangerous (because of various R quirks) operations that you'll perform with the language. Knowing what the dangers are and to expect them rather than be surprised by them is a defensive strategy, and there's value in understanding what's going on. Of course, R being an extensible language means that it only takes a bit more effort to replace the "bad" bits with something that makes more sense — and sure enough, that's where we're leading.

### **6.1 Text processing**

Joining together pieces of text is only one of the ways we can

interact with strings. Often we need to extract parts of a string or replace one part with something else. This comes about most frequently when data is not quite of the form we require or has been formatted in an odd way. Being able to identify particular patterns in text also means we can replace those wherever they appear. Once you begin working with thousands of strings at once, you'll be thankful you don't have to do any of the work manually.

### 6.1.1 TEXT MATCHING

Because strings are stored as collections of characters, we can think of them in that way and perform *comparisons* (covered in section 3.4) between groups of characters and particular letters or patterns. One of the simplest questions to ask of a string is, “does it start with...?” Let’s check this with the base function `startsWith()`:

```
startsWith(x = "banana", prefix = "b")
#> [1] TRUE
```

There is a complementary function for checking the end of a string:

```
endsWith(x = "apple", suffix = "e")
#> [1] TRUE
```

Note that this won’t try to do anything clever like ignore whitespace. If a string starts with a space, then it starts with a space, not a letter:

```
startsWith(" banana", "b")      ①
#> [1] FALSE
```

①

You can drop the argument names once you’re familiar with the ordering.

Uppercase and lowercase are also considered distinct:

```
startsWith("Apple", "a")
```

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

The prefix or suffix you're checking can be any length — it need not be a single letter:

```
startsWith("carpet", "car")
#> [1] TRUE
```

Searching the start and end of a string are two specific cases of what you may want to do more generally: search anywhere in the string. In that case, you require a more advanced tool. If you're already familiar with Unix/Linux command-line tools, you may recognize the `grep()` function, which finds text that matches a pattern. Given a pattern, a text string, and the argument `value = TRUE`, `grep()` will return any text strings (the entire string) that contain the pattern:

```
grep(
  pattern = "nan",
  x        = "banana",
  value    = TRUE  ①
)
#> [1] "banana"
```

①

Returns the matching strings if `value = TRUE`; otherwise returns the vector indices of strings that contain the pattern

In a similar fashion to the `startsWith()` and `endsWith()` functions, you can produce a logical result that indicates whether or not the pattern can be found in a string using the `grep1()` (1 for logical) function:

```
grep1("app", "apple")
#> [1] TRUE
```

It's worth noting that all of the preceding functions are vectorized, so you can compare and process entire vectors of strings at once:

```
grep1(pattern = "meow", x = c("meowing", "meowed",
"homeowner"))
```

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

In these cases, with the `value = TRUE` argument, `grep()` returns only the strings that contain the pattern:

```
grep("car", c("cradle", "cartoon", "carpet"), value = TRUE)
#> [1] "cartoon" "carpet"
```

Without this argument (or `value = FALSE`, the default), it returns the index corresponding to strings that contain the pattern (with only a single string, this is either `1` for a match, or `integer(0)` for no matches):

```
grep("car", c("cradle", "scary", "carpet", "Carroll"))
#> [1] 2 3
```

These patterns are overly simple, and you'll rarely require them. These functions will become vastly more useful shortly when we expand on how the patterns can be structured more generally.

## A better way

As you may have guessed, there are more sophisticated third-party ways of achieving these tasks. The `stringr` package (another of Hadley Wickham's) provides simple tools to replace most of the functions covered in this section:

```
# install.packages("stringr")
library(stringr)
```

For matching strings to patterns, the `str_detect()` function essentially replaces `grep1()`, and `str_view()` provides a very neat visualization (see the figure) of which strings have matches (and where) inside the RStudio Viewer pane:

```
# install.packages("htmlwidgets")
str_view(c("cradle", "scary", "carpet", "Carroll"),
"car")
```

```
cradle
scary
carpet
Carroll
```

### Example of `str_view()` in the Viewer pane

`str_view()` will show only the first match, whereas `str_view_all()` will show all matches in the case where there are multiple matches within a string.

**NOTE** The particular function I'm demonstrating here *also* requires the `htmlwidgets` library and its dependencies in order to show a HTML rendering of the string matches. It's not a formal dependency of `stringr`, so it doesn't automatically get installed. To use this, you'll need to install `htmlwidgets` yourself.

## 6.1.2 SUBSTRINGS

The pattern-matching functions discussed so far are useful for determining *which* strings contain a pattern, but you're still left with the strings as a whole. If you wish to isolate part of a string, there are plentiful options to do so.

The `substring()` and `substr()` functions at first glance appear to do exactly the same thing: isolate a substring. They're subtly different, though, and are capable of much more. Each takes a vector of strings (which can be just one, of course), and two positions: the `start/first` and `stop/last` indices of the substring to extract. The most visible difference between these functions is that for `substring()`, the default of `last = 10000000L` means you may only need to specify the `first` argument to extract text from that position to what will surely be the end of your strings.

For example, to extract the first letters of some strings, you could use the following:

```
substr(x = c("You", "Only", "Live", "Once"), start = 1,
stop = 1)
#> [1] "Y" "O" "L" "O"
```

Note that I've specified both the `start` and `stop` values as `1` since I don't want to extract past that value. If you wanted to extract the first three letters of a word, you'd use this:

```
substr("carpet", 1, 3)
#> [1] "car"
```

You could save a tiny amount of writing if you wanted to extract the *last* three letters, by using the `substring()` function (with its default `last` argument):

```
substring("carpet", 4)
#> [1] "pet"
```

These examples are only useful if you already know which indices you require for extracting a substring. Often that's not the case, so let's turn to more advanced functions.

**TIP** Again, the `stringr` package provides better ways to extract substrings. One option is the `str_extract()` function. Until we have better tools to work with the patterns, though, all we can do is extract the part we already know: the pattern.

### 6.1.3 TEXT SUBSTITUTIONS

Replacing a substring is a common task, and there are many ways of doing it. The simplest is the `sub()` function. This is structured similar to the `grep()` function except there's also a replacement argument. You can replace a pattern with some other text (which need not be the same number of characters):

```
sub(pattern = "pet", replacement = "tography", x =
"carpet")
#> [1] "cartography"
```

This can be particularly useful for inserting things that are

missing or incorrectly formatted:

```
sub(" 1", " L", "the Tower of london is located on the bank  
of the Thames.") ①  
#> [1] "the Tower of London is located on the bank of the  
Thames."
```

①

**Including a space in the pattern (“ l”) only matches the start of words in the middle of a sentence.**

Note that `sub()` will only replace the first match to the pattern, despite both `london` and `located` being matches. To replace *all* matches, you need the `gsub()` function:

```
gsub("mrs", "Mrs", "mrs smith, mrs jones, and mrs martin.")  
#> [1] "Mrs smith, Mrs jones, and Mrs martin."
```

This can be useful for altering unwanted formatting, provided you remember to *escape* control characters inside argument strings (or use different characters):

```
gsub("\!", "\\"", "He said 'Hello'")  
#> [1] "He said \"Hello\""
```

**TIP** The `stringr` package provides `str_replace()` and `str_replace_all()` for the single and multiple replacement scenarios.

This is also how to deal with the “commas in numbers” issue noted back in section 2.1.1 — replace commas with nothing (an empty string, ""):

```
str_replace_all("1,200,000", ",","", "")  
#> [1] "1200000"
```

## 6.1.4 REGULAR EXPRESSIONS

The string patterns you’ve used so far have been simple strings themselves, but it’s vastly more likely that you’ll need to replace

some group of letters, numbers, or symbols with certain characteristics. To do that, you must be able to refer to these groups using text, using *regular expressions* or *regex*.

**DEFINITION** *Regular expression (regex)* refers to a pattern specified by a system of (possibly escaped) characters in order to capture text matches.

This topic is somewhat advanced, but it will be useful enough for you to be able to recognize when you need regular expressions, and I'll show you some easy examples. You can extract or replace any single digit in a string with individual replacements easily enough if you know which digit you're looking for:

```
sub(pattern = "3", replacement = "z", x = "xy3")
#> [1] "xyz"
```

But what if there's a general case where the digit could be any of 0, 1, ..., 9? In that case, use a *regular expression* that matches to *any* digit. There are several ways to specify this: using \d (an escaped d, discussed in section 2.1.2, where the backslash itself will need to be escaped), or using the grouping of digits [0123456789] (square brackets in a regular expression means “any of these”), using a shorthand notation for that group [0-9], or the named digit class [:digit:]). Even if you don't know exactly which digit you need to replace, the following will still work fine:

```
sub(pattern = "\d", replacement = "z", x = "xy8")      ①
#> [1] "xyz"
```

①

The backslash itself needs to be escaped here.

A detailed breakdown of regular-expression selectors is available in the help menu for ?`regex`, and a brief list of some common examples is shown in table 6.1.

**Table 6.1** Common regex selectors

To select...	Use...	Examples using the string “xY8 .”
A specific letter\number	The letter\number, such as x	Matches x
Digits	\d, [0-9], [:digit:]	Matches 8
Uppercase	[A-Z], [:upper:]	Matches Y
Lowercase	[a-z], [:lower:]	Matches x
Space	\s, [:space:]	Matches the space between 8 and .

These can be woven together with the parts you do know to build up a full *regular expression*, and that can be used in functions such as `grep()` and `sub()`. For example, to select a c or C (starting from anywhere in the string) followed by ar you could use the regular expression "[Cc]ar":

```
grep("[Cc]ar", c("cradle", "scary", "carpet", "Carroll"),
value = TRUE) ①
#> [1] "scary"  "carpet"  "Carroll"
```

①

**[Cc]** selects either C or c.

There are also ways to select the start (^) and end (\$) of a string, boundaries between words and spaces (\b), or any character at all (.). On top of all that, there are quantifiers that can follow a pattern to select how many times that pattern may appear in a valid match. These are briefly summarized in table 6.2.

**Table 6.2** Common regex quantifiers for selecting multiple pattern matches

To select a pattern ... times	Use..	Examples
	.	

0 or 1	?	\d? matches xyz and xy8z
0 or more	*	\d* matches xyz and xy8z and xy89z
1 or more	+	\d+ matches xy8z and xy89z
n	{n}	\d{3} matches xy789z

You can use all this in previously used functions to build very specific expressions:

```
grep("[Cc]ar{2}", c("cradle", "scary", "carpet",
"Carroll"), value = TRUE) ①
#> [1] "Carroll"
```

①

r{2} selects two rs. Note that the entire expression must match.

There are many more options available to specify a regular expression, and hopefully it's apparent how powerful they can be in the right hands. With that flexibility comes complexity, so it's understandable that these aren't used to their full potential often enough.

To ease the difficulty of specifying the correct regular expression for a match, there are several tools that can help. The first is the `stringr` function `str_view()`, mentioned earlier, which provides a visualization of the matches in the RStudio Viewer pane. Given multiple matches, you can use the `str_view_all()` function. This is very handy for testing out regular expressions. For example, matching two digits can be achieved with the following:

```
str_view_all("xy23yx878xyx9yxy", "\d{2}") ①
```

①

Matches two digits in a row

The result is shown in figure 6.1.

**Figure 6.1** Matching two digits in a row using `str_view()`, shown in the Viewer pane

The `rex` package helps to build the regular expressions by providing some shortcuts using more R-like syntax:

```
# install.packages("rex")
library(rex)
#
#> Attaching package: 'rex'
#> The following object is masked from 'package:stringr':
#>
#>     regex
```

This makes building up a complex regular expression much cleaner and easier to read. To select the start of a string followed by any three letters regardless of case, one digit, and then any number of lowercase letters, you can build this:

```
matching_regex <- rex(
  start,
  n_times(letter, 3),
  n_times(digit, 1),
  any_lowers
)
matching_regex
#> ^(?:[:alpha:]){3}(?:[:digit:]){1}[:lower:]*
```

The `(?: )` wrappers around each component have additional functionality that you're not making use of here, but that's how the `rex` package builds the expression. You can find matches to some strings with `str_view()`:

```
possible_strings <- c("xyzabc9?AB", "JDC8xyzabc", "SC?
0abcxyz")
str_view(possible_strings, matching_regex)
```

The result is shown in [figure 6.2](#).

```

xyzabc9?AB
JDC8xyzabc
SC?0abcxyz

```

**Figure 6.2 Using the `rex` package to build up a regular expression that can be used in `str_view()`**

Once you become familiar with regular expressions, you'll find they become extremely powerful when working with text. Until now, though, you've been selecting parts of strings from either one or several strings. To be able to work with data in general, you'll need to be able to do similar things with the data structures you create.

## 6.2 *Selecting components from structures*

Different structures require different commands to extract specific components, and for various reasons commands behave differently depending on how you're using them. A lot of this stems from trying to make life easier, but with any guesses about what someone is trying to achieve, inevitably structural consistency is the cost.

### 6.2.1 VECTORS

If you have a vector of values and want to extract just some of them, you select them by their index — their position in the vector. For example, say you have this vector:

```
vec <- c(21, 23, 25, 27, 29)
```

To extract just the second element, use the square-bracket operator (yes, a function under the hood, most frequently referred to by just [ even though R requires the corresponding ] to be present too) with the desired indices:

```
vec[2]
#> [1] 23
```

**NOTE** You may have heard that other programming languages (such as C and Python) start counting elements at 0.<sup>1</sup> Not R, which has the same numbering style as Fortran (a language at least a decade older than C) and starts counting as in everyday conversation: the first element is number 1, the second is number 2, and so on.

---

<sup>1</sup> So that elements can be referenced in memory using an offset; the first element has offset 0, the next has offset 1, and so on.

---

This operator can itself take a vector of values, so you can select multiple values at once:

```
vec[c(2, 5)]
#> [1] 23 29
```

A sequence is just a vector, so you can extract a run of values all at once:

```
vec[2:4]
#> [1] 23 25 27
```

To extract everything *except* a particular index, prepend that index with a minus sign:

```
vec[-2]
#> [1] 21 25 27 29
```

Or exclude a vector or indices:

```
vec[-c(2, 3)]
#> [1] 21 27 29
```

If you supply no indices, you get the entire vector

```
vec[]
#> [1] 21 23 25 27 29
```

although this is equivalent to just requesting the vector `vec`.

The vector doesn't have to be saved in a variable to use this;

you can extract values from a vector you just defined:

```
c(32, 34, 36, 38)[c(2, 3)]
#> [1] 34 36
```

There's a second form of this operator named `[[]]`. The object to be subset is followed by `[[]`, then the subsetting details, and then `]]`, but this combination is most frequently referred to by the function name `[[]]`. R knows that the corresponding closing `]]` will need to be present for the expression to make sense. This can only select a single element from a vector. Attempting to select more than one produces an error:

```
vec[[c(1, 2)]]
#> Error: attempt to select more than one element in
vectorIndex
```

One advantage of this, though, is that it removes names from vectors. Suppose you have a named vector:

```
named_vec <- c(x = 21, y = 23, z = 25)
```

You can extract a single element with its name using

```
named_vec[2]
#> y
#> 23
```

or without its name:

```
named_vec[[2]]
#> [1] 23
```

There are few occasions when you'll need to extract a single element from a vector this way, but it's a different story with more complex objects.

Identifying elements to be extracted based on their position (index) is fragile and should generally be avoided; the ordering may change, elements added or removed, and so on, so it's unreliable to depend on these for extraction (though sometimes that's all you have). A more robust way to extract named

element(s) is by combining [ or [[ with their name(s). To extract a named element from a vector, pass the name of the element(s) into [ or [[ as a character vector:

```
named_vec[c("x", "y")]
#> x y
#> 21 23
```

Here's another example:

```
named_vec[["y"]]
#> [1] 23
```

Again, [[ drops the names and only returns single elements.

## Unexpected selections

A surprising feature of index-based extraction is that using non-integers does work. If you subset the built-in English alphabet vector `letters` by non-integer values

```
letters[c(1.3, 1.5, 1.7, 2.1)]
#> [1] "a" "a" "a" "b"
```

you extract the first element three times and the next one once.

In these cases, the non-integer indices are *truncated* to just the integer part of the number (discussed in section 2.3) that `as.integer()` rounds toward 0, and then the extraction is performed. This perhaps isn't so surprising; the selections shown earlier used non-integer (strictly speaking) index specifications; for example, in `vec[c(2, 5)]` you haven't used `2L` and `5L`.

Section 5.1 explains that in R, indexes start at 1. Trying to select the 0 element doesn't produce an error or a warning, but the element certainly doesn't exist:

```
letters[0]
#> character(0)
```

Instead you've produced an *empty* result of type *character* (because `letters` is a character vector). Trying to select elements with an index greater than the length of a vector results in an `NA` value:

```
letters[99]
#> [1] NA
```

Attempting to extract the element with index `NA` leads (as you may expect) to `NA`, but (as you may not expect), to not just one of them:

```
named_vec[NA]
#> <NA> <NA> <NA>
#>     NA     NA     NA
```

`NA` (without being more specific) is actually a logical type, and as you'll see, you can select elements with logical operators. Any comparison involving `NA` results in `NA`, so every element matches this criteria. You get just a single `NA` if you use a more specific missing value, such as the character version of `NA`:

```
named_vec[NA_character_]
#> <NA>
#>     NA
```

Take some time to practice these extraction techniques, because they're fundamental to further processing.

## 6.2.2 LISTS

To extract an element from a list, use the same syntax as with vectors. Given a list with several elements

```
lst <- list(
  odd    = c(1, 3, 5),
  even   = c(2, 4, 6),
  every  = 1:6
)
```

you can extract one or more of the elements with the `[` operator, which retains their names:

```
lst[c(1, 2)]
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 2 4 6
```

Note that this returns the first and second *vectors*, but the result is still a *list*:

```
typeof(lst[c(1, 2)])
#> [1] "list"
```

To extract elements of those, first specify *which* vector to extract from (selecting only one). Trying to do that with `[` still returns a list:

```
typeof(lst[c(1, 2)][2])
#> [1] "list"
```

Recall that the double operator `[[` returns only a single element and drops its name. It also returns a vector from a list:

```
lst[[2]]
#> [1] 2 4 6
```

```
typeof(lst[[2]])
#> [1] "double"
```

You can extract elements from this easily, since this is a vector:

```
lst[[2]][2]
#> [1] 4
```

If you find yourself in the unfortunate position of dealing with *nested* lists (lists inside lists), then you may need to specify multiple values to `[[` in order to specify a nested element. For example, the following nested list is valid, where the first element of the list is a list:

```
nested_lst <- list(
  names = list(
```

```

    first = "Jonathan",
    last = "Carroll"
),
fav_lang = "R"
)
nested_lst
#> $names
#> $names$first
#> [1] "Jonathan"
#>
#> $names$last
#> [1] "Carroll"
#>
#>
#> $fav_lang
#> [1] "R"

```

To extract the second element from the inner list, use the following:

```

nested_lst[[c(1, 2)]]
#> [1] "Carroll"

```

You can extract named elements by specifying their name(s):

```

lst[c("even", "odd")]
#> $even
#> [1] 2 4 6
#>
#> $odd
#> [1] 1 3 5

```

Here's another example:

```

lst[["even"]]
#> [1] 2 4 6

```

This helps somewhat with nested lists, since keeping track of the position indices gets confusing:

```

nested_lst[[c("names", "last")]]
#> [1] "Carroll"

```

In addition to a name in [[, though, you can use the name to extract a single element using a dollar symbol \$ between the name of the list and the name of the element:

```
lst$odd
#> [1] 1 3 5
```

Note that this doesn't work for vectors, even if their elements are named:

```
named_vec$a
#> Error: $ operator is invalid for atomic vectors
```

Be very careful with this feature, because R performs *partial matching* when it uses this dollar-name syntax, as it does for arguments (discussed in section 4.1.7). You don't necessarily have to provide the *entire* name of the element to extract (although you should), just enough of it so that it can be uniquely determined:

```
lst$o
#> [1] 1 3 5
```

This becomes an issue if you expect this to definitely work. There may *not* be enough of the partial name to uniquely determine the element:

```
lst$e
#> NULL
```

Note that the result is the same if the column doesn't exist:

```
lst$z
#> NULL
```

To be notified when this partial matching occurs (with a warning), then the option can be turned on by evaluating the following at the start of the session

```
options(warnPartialMatchDollar = TRUE)
```

which produces

```
lst$o
# Warning message:
# In lst$o: partial match of 'o' to 'odd'
#> [1] 1 3 5
```

The partial matching of names does not occur for the more robust name-extraction method

```
lst[["o"]]
#> NULL
```

unless you explicitly ask for it with the `exact` argument:

```
lst[["o", exact = FALSE]]
#> [1] 1 3 5
```

This option can be `TRUE` (only allow exact matches), `FALSE` (silently allow partial matches), or `NA` (allow partial matches but provide a warning when used). Best practice is to not rely on a fragile feature and to supply the full name when using either convention.

Because you can specify `[[]` extraction with names, that makes accessing the vector elements appear a lot cleaner

```
lst$odd[2]
#> [1] 3
```

even with nested lists:

```
nested_lst$names$last
#> [1] "Carroll"
```

## “Object of type ‘closure’ is not subsettable”

One of the most frequent trip-ups people encounter is a seemingly obscure reference to something not being *subsettable*. This comes about with functions — for example, this one that creates a list with three values:

```
make_a_list <- function() {
  return(list(x = 11, y = 12, z = 13))
}
```

You can evaluate this and generate the list as an object:

```
new_list <- make_a_list()
```

You may want to extract one of the elements, in which case you extract from this new variable:

```
new_list$y  
#> [1] 12
```

Or you can save the trouble of naming that object and extract it straight from the expression that generates it:

```
make_a_list()$y  
#> [1] 12
```

This is valid because R will break it down into the relevant steps before doing any of the operations; the function will be evaluated, and *then* the subsetting operator \$ will be applied.

But if you forget the parentheses that identify that you're calling the function `make_a_list()`, R will treat this as trying to subset the *function* (which is stored as an object). That's not allowed, though, so it fails:

```
make_a_list$y  
#> Error: object of type 'closure' is not subscriptable
```

The error message makes sense if you know that a *closure* is a proper name for a function object (which encloses both a series of expressions and the environment from which it was created).

### 6.2.3 MATRICES

When you have a structure with more than one dimension, such as `matrix`, specifying the indices to extract requires two values: one for the row and one for the column. This is still handled by the `[` operator, but now it includes a comma (,) between the row and column indices. Given a matrix consisting of the first 12 letters of the alphabet (constructed using `[` subsetting on the

built-in letters vector)

```
mat <- matrix(letters[1:12], nrow = 3)
mat
#>      [,1] [,2] [,3] [,4]
#> [1,] "a"  "d"  "g"  "j"
#> [2,] "b"  "e"  "h"  "k"
#> [3,] "c"  "f"  "i"  "l"
```

you can extract a single element by specifying the row and column indices with the structure `m[row, col]`, so the element in the third row and second column is extracted as follows:

```
mat[3, 2]
#> [1] "f"
```

Multiple elements can be extracted by supplying a vector to either the row or column position (or both) — for example, extracting the first *and* third rows and second column:

```
mat[c(1, 3), 2]
#> [1] "d" "f"
```

Note that in each of these cases, a vector is returned. This is again R being more helpful than it's asked to be. One of the arguments to `[` is `drop`, and it defaults to `TRUE`. The consequence is that whenever the extraction result can be converted to a single dimension (a vector), it will be. Instead of returning a single-column or single-row matrix, R returns a vector. That may be what you really want to work with, but it's an inconsistency that makes programming difficult.

**CAUTION** It's worth noting that in these examples the argument names are left out and just assumed by position. `[` has formal arguments `i` and `j`, which in theory would select the row (`i`) and column (`j`) elements of a matrix. In practice, though, these explicit arguments are ignored, and the selection is performed by position. Trying to swap these arguments around and extract with `mat[j = 2, i = 3]` will produce the same as `mat[2, 3]`.

If you select enough dimensions such that the result can't be reduced to just a vector, a matrix is returned:

```
mat[c(1, 3), c(2, 4)]
#>      [,1] [,2]
#> [1,] "d"  "j"
#> [2,] "f"  "l"
```

**NOTE** Some confusion may arise because you asked for the first and third rows as well as the second and fourth columns, but the resulting matrix has row and column labels with 1 and 2 only. These aren't “names” as such; they're merely indices of the *new* matrix. Because the new matrix has two rows and two columns, the indices cover this range.

The result of extracting part of a matrix will always be rectangular, having values in all rows and columns;<sup>2</sup> there is no way to create a “triangular” matrix without the rest of the matrix being `NA`. This means that the resulting object (either a matrix or a vector if `drop = TRUE`) will be formed by the intersection of the requested rows and columns. You won't produce a matrix with blank elements. But you can specify that you want *all* elements in a row or column by leaving out that selection (a space or merely nothing before or after the comma).

---

<sup>2</sup> “Square” matrices or single values are specific cases of “rectangular.”

To subset to all rows, leave out the row selection:

```
mat[ , c(2, 3)] ①
#>      [,1] [,2]
#> [1,] "d"  "g"
#> [2,] "e"  "h"
#> [3,] "f"  "i"
```

①

The space before the comma isn't mandatory, but it can make the selection clearer.

When a matrix has names, such as the following

```
colnames(mat) <- c("col1", "col2", "col3", "col4")
rownames(mat) <- c("row1", "row2", "row3")
mat
#>      col1 col2 col3 col4
#> row1 "a"  "d"  "g"  "j"
#> row2 "b"  "e"  "h"  "k"
#> row3 "c"  "f"  "i"  "l"
```

you can use these names to perform the extraction:

```
mat[c("row1", "row3"), c("col2", "col4")]
#>      col2 col4
#> row1 "d"  "j"
#> row3 "f"  "l"
```

You can also use the double [ [ syntax as long as you're extracting a single element:

```
mat[["row2", "col2"]]
#> [1] "e"
```

## Escaping naming rules with quotes or backticks

Each of the named-object extractions has used either the explicit square-bracket syntax that takes the name of the element(s) to extract, or the dollar-name (\$) shortcut with the bare name of the element(s).

A complication of this is when names bypass the usual rules and include spaces, start with numbers, and so on. These names need to be surrounded by quotes or backticks, but it otherwise works similarly:

```
c(x = 7, "y variable" = 8, z = 9)["y variable"]
#> y variable
#>     8
```

```
list(x = 7, ".2b" = 8, z = 9)$`.2b`
#> [1] 8
```

## 6.3 Replacing values

Extracting a component of a structure is great if you want to isolate that, but sometimes you need to *change* a part of a structure. This requires selecting the part to update (using the indexing methods we've just covered). Typically this involves those same steps along with an assignment operator (`<-`). Most of the extraction operators provide this additional functionality in the form of another function. For `[`, there is a replacement operator (formally: `[<-]`), which R translates to when it sees a subsetted object being assigned a new value.

If you have a vector `vec` and you want not to extract the second element, but to *modify* it, you can do so with the following:

```
vec[2] <- 99
vec
#> [1] 21 99 25 27 29
```

Note that evaluating the assignment operator still produces no output in the Console, so remember to `print()` (or evaluate) the object if you want to inspect the changes.<sup>3</sup>

---

<sup>3</sup> As a reminder, surrounding a non-printing expression with parentheses will force it to print, such as `(no_print <-2)`.

The new addition may cause the *type* of the object to be coerced:

```
vec[2] <- "99"
vec
#> [1] "21" "99" "25" "27" "29"
```

An element of the matrix can be updated similarly:

```
mat[3, 2] <- 99      ①
mat
#>      col1 col2 col3 col4
#> row1 "a"  "d"  "g"  "j"
#> row2 "b"  "e"  "h"  "k"
#> row3 "c"  "99" "i"  "l"
```

(1)

**The new element is a numeric type, but it will be coerced to character since that's the most general type of the two.**

The replacements can also be *new* elements that previously didn't exist, with some caution over how these are produced. Given a vector with length 5

```
vec <- 1:5
vec
#> [1] 1 2 3 4 5
```

you can request the eighth value

```
vec[8]
#> [1] NA
```

even though it isn't defined (and so is set to NA). You can *replace* (create) the eighth element, as long as you also create the sixth and seventh as well. You don't get a choice in this matter; they're automatically created and set to NA:

```
vec[8] <- 99
vec
#> [1] 1 2 3 4 5 NA NA 99
```

For a matrix like the following

```
mat <- matrix(1:8, nrow = 2)
mat
#>      [,1] [,2] [,3] [,4]
#> [1,]     1     3     5     7
#> [2,]     2     4     6     8
```

this isn't allowed:

```
mat[8, 2] <- 99
#> Error: subscript out of bounds
```

You can replace several values at once using more vectors, as long as you're careful about matching the lengths of the elements being replaced, and the lengths of their replacements

```
vec <- 1:8
vec[2:3] <- c(98, 99)
vec
#> [1] 1 98 99 4 5 6 7 8
```

allowing still for the case where recycling works:

```
vec[1:7] <- 0
vec
#> [1] 0 0 0 0 0 0 0 8
```

If you don't want to permanently overwrite the vector, there is a handy function in the base package, `replace()`, which does what the name suggests (without assigning the result) — it takes a vector  $x$  and replaces the elements at positions  $list$  with values (recycling if necessary). You can replace the odd elements of  $1:6$  with 9:

```
replace(x = 1:6, list = c(1, 3, 5), values = 9)
#> [1] 9 2 9 4 9 6
```

Similarly for a matrix, you can replace an entire column at once:

```
mat[, 2] <- c(98, 99)
mat
#> [,1] [,2] [,3] [,4]
#> [1,] 1 98 5 7
#> [2,] 2 99 6 8
```

For lists, you can request the second element with `[[2]]`:

```
lst <- list(
  odd = c(1, 3, 5),
  even = c(2, 4, 6),
  every = 1:6
)
lst[[2]]
#> [1] 2 4 6
```

You can update that element in the same way as with vectors

```
lst[[2]] <- c(8, 10)
lst
#> $odd
#> [1] 1 3 5
#>
```

```
#> $even
#> [1] 8 10
#>
#> $every
#> [1] 1 2 3 4 5 6
```

or with the dollar-name syntax:

```
lst$even <- c(10, 12)
lst
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 10 12
#>
#> $every
#> [1] 1 2 3 4 5 6
```

You can even update single elements within those levels:

```
lst$even[2] <- 100
lst
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 10 100
#>
#> $every
#> [1] 1 2 3 4 5 6
```

To *remove* elements from a list, either deselect them with a minus sign

```
lst <- lst[-2]
lst
#> $odd
#> [1] 1 3 5
#>
#> $every
#> [1] 1 2 3 4 5 6
```

or set their value to `NULL`:

```
lst$every <- NULL
lst
```

```
#> $odd
#> [1] 1 3 5
```

Attempting to add additional elements behaves similarly to vectors

```
lst[[4]] <- c("x", "y", "z")
lst
#> $odd
#> [1] 1 3 5
#>
#> [[2]]
#> NULL
#>
#> [[3]]
#> NULL
#>
#> [[4]]
#> [1] "x" "y" "z"
```

except that no placeholder value is created automatically; the intermediate elements are given the value `NULL`. Note that the new unnamed element (and automatically created elements with no names) are designated by their position in the list, whereas the named elements have their names.

You can similarly add more elements without worrying about how long the list already is by using the dollar name syntax:

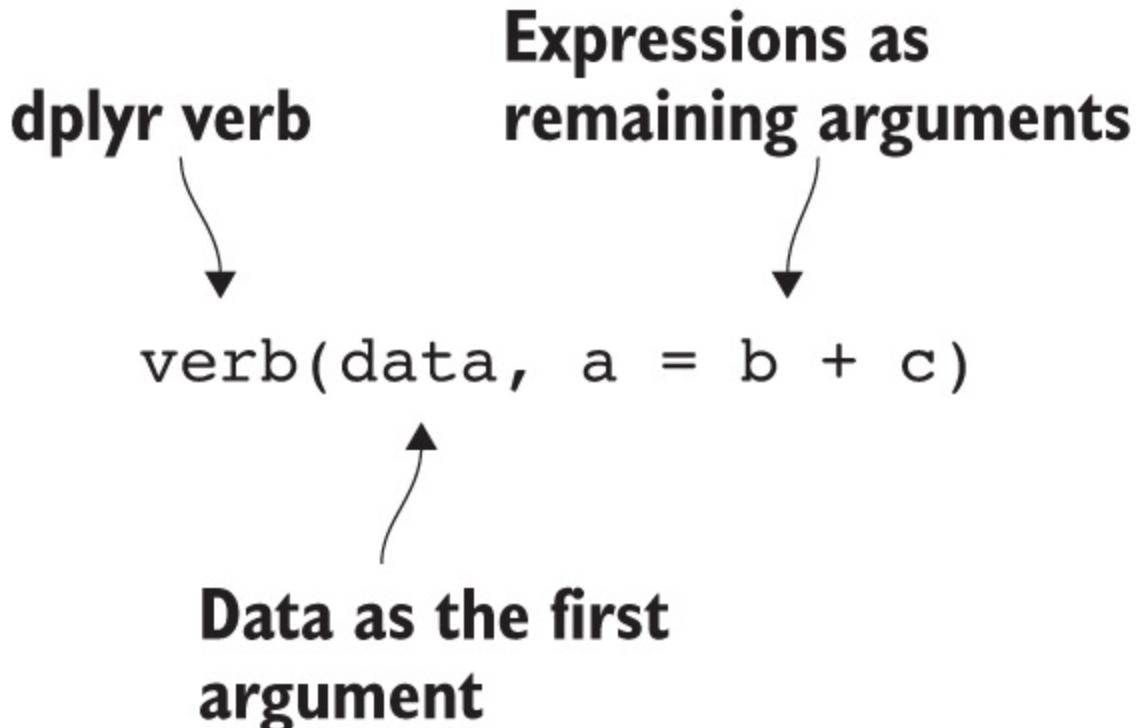
```
lst$greek <- c("alpha", "beta", "gamma")
lst
#> $odd
#> [1] 1 3 5
#>
#> [[2]]
#> NULL
#>
#> [[3]]
#> NULL
#>
#> [[4]]
#> [1] "x" "y" "z"
#>
#> $greek
#> [1] "alpha" "beta" "gamma"
```

**NOTE** I've said from the start that R doesn't modify data, yet we've just gone through many examples that appear to do exactly that. What's happening here is that R is making a copy of an object, modifying *that*, and then saving it back to the original variable name. This can use up a lot of resources if the objects are large, so don't be surprised if you're trying to update the 13 millionth element of a `data.frame` column and R is taking a while. If you really need things done faster, the `data.table` package modifies objects *in memory* rather than making copies of them.

## 6.4 `data.frames` and `dplyr`

When performing selections, extractions, and replacements on `data.frames`, the previously discussed mechanisms extend in more or less expected ways, with a whole slew of caveats and potential trip-ups. `$` or `[]` can be used to extract an entire column (because a `data.frame` is a list of vectors), or `[` can be used to extract elements in the same way as for a matrix. There is, however, a better way.

The `dplyr` (“dee-ply-er”) package by Hadley Wickham makes working with `data.frames` as simple as possible through a consistent and well-thought-out set of *verbs* (see figure 6.3).



**Figure 6.3** `dplyr` verb construction

`dplyr` is at the top of my list of recommended add-on packages for R. I start every session with it loaded because I'm certain I'm going to use it. It provides a consistent interface of verbs for manipulating *tidy data* — data that is formatted in a proper, regulated manner, with one row per observation and one column per quantity. This is the structure of a `data.frame`, hence the `d` in `dplyr`. The functions in this package take one or more `data.frames` and produce modified copies of it/them with rows and/or columns selected, added, removed, or altered in some way.

In order to enable the use of this, you need to install, load, and attach the package. Installation of this package can take quite a while since much of it is written in more memory-efficient C code, which requires compiling:

```
# install.packages("dplyr")
library(dplyr)
```

Now that the functions are available, it's time to learn some of the most important ones. These are the superstars of the `dplyr`

package and feature in the vast majority of analyses I've created. They're carefully constructed in the “correct” style of writing a function — the short name describes what action the function performs with a *verb* (a “doing” word) and returns a predictable *type* of object. Another feature of these verbs is that they consistently have `.data` as their first argument and ... as the remaining argument that captures expressions to be processed.

### 6.4.1 DPLYR VERBS

The most important `dplyr` verbs you’ll want to become acquainted with — which achieve the same goals as the above manipulations of smaller structures — are as follows:

- `mutate` — Add or alter columns, defined by some calculation or operation on existing columns. Compares to selecting a subset of data and assigning new values to it, either a new column or an existing one:

```
head( ①
  mutate(mtcars, displ_1 = disp / 61.0237)
)
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
displ_1
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4    4
2.621932
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4    4
2.621932
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4    1
1.769804
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3    1
4.227866
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3    2
5.899347
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3    1
3.687092
```

①

Only prints the first six rows

- `select` — Keep only these columns. The order in which the selected variables are provided will be the new ordering of the

columns. Compares to named subsetting of columns. Attempting to select a column that doesn't exist in the data will produce an error. Helper functions also exist to select columns that match criteria, such as `starts_with()`, `contains()`, and `everything()`. See `?select` for more:

```
head(
  select(mtcars, cyl, mpg)
)
#>          cyl  mpg
#> Mazda RX4      6 21.0
#> Mazda RX4 Wag   6 21.0
#> Datsun 710      4 22.8
#> Hornet 4 Drive   6 21.4
#> Hornet Sportabout 8 18.7
#> Valiant         6 18.1
```

- `filter` — Keep only rows matching some logical criteria. These criteria can be passed in one at a time or joined with a logical operator (`and`: `&`, `or`: `|`). Compares to named subsetting of rows:

```
filter(mtcars, cyl < 6, am == 1)
#>   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> 1 22.8   4 108.0 93 3.85 2.320 18.61  1  1     4    1
#> 2 32.4   4  78.7 66 4.08 2.200 19.47  1  1     4    1
#> 3 30.4   4  75.7 52 4.93 1.615 18.52  1  1     4    2
#> 4 33.9   4  71.1 65 4.22 1.835 19.90  1  1     4    1
#> 5 27.3   4  79.0 66 4.08 1.935 18.90  1  1     4    1
#> 6 26.0   4 120.3 91 4.43 2.140 16.70  0  1     5    2
#> 7 30.4   4  95.1 113 3.77 1.513 16.90  1  1     5    2
#> 8 21.4   4 121.0 109 4.11 2.780 18.60  1  1     4    2

## or equivalently filter(mtcars, cyl < 6 & am == 1)
```

- `summarize`(*also summarise*) — Aggregate multiple values together and provide them to a function, returning a single result. This result will still be a `data.frame`, so further processing may be required if you only want a value. Compares to the `stats` function `aggregate()`, particularly when the input is a grouped `tbl_df`:

```
summarize(mtcars, mean(disp)) ①
#>   mean(disp)
#> 1  230.7219
```

(1)

**Calculates the mean of disp, taking into account any groups by which the data has been split**

- group\_by — Apply a grouping for which operations will be performed by group. Compares to the internals of aggregate(), which splits a data.frame by some grouping variable before computing the summary function:

```
by_cyl <- group_by(mtcars, cyl)      ①
head(by_cyl)
#> # A tibble: 6 x 11
#> # Groups:   cyl [3]
#>   mpg   cyl  disp    hp  drat    wt  qsec    vs    am
#>   gear  carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>   <dbl> <dbl>
#> 1 21.0  6.00   160 110     3.90  2.62  16.5  0     1.00
#> 2 21.0  6.00   160 110     3.90  2.88  17.0  0     1.00
#> 3 22.8  4.00   108 93.0    3.85  2.32  18.6  1.00  1.00
#> 4 21.4  6.00   258 110     3.08  3.22  19.4  1.00  0
#> 5 18.7  8.00   360 175     3.15  3.44  17.0  0     0
#> 6 18.1  6.00   225 105     2.76  3.46  20.2  1.00  0
#> 7 14.3  8.00   360 175     3.08  4.08  17.8  0     0
```

(1)

**Groups the mtcars data by number of cylinders**

Notice that the output printed for the result of the group\_by operation looks different: data.frame doesn't support grouping like this, so this result is a tibble (discussed in section 5.7) and uses print.tbl\_df for printing.

Groupings persist and are recognized by other verbs, so you can summarize() a grouped object to produce a grouped summary:

```
summarize(by_cyl, disp_bar = mean(disp), hp_bar =
```

```
mean(hp)      ①
#> # A tibble: 3 x 3
#>   cyl disp_bar hp_bar
#>   <dbl>    <dbl>   <dbl>
#> 1 4.00     105    82.6
#> 2 6.00     183    122
#> 3 8.00     353    209
```

(1)

**For each group, calculates the mean of the disp and hp columns**

Many more useful functions are provided by the `dplyr` package, which I encourage you to discover. For now, these will provide ample opportunity to create rich analyses from your data in a clean and reliable manner.

**NOTE** These verbs don't alter the original input data (in these examples, the `mtcars` dataset) but rather return a new copy with the relevant operation performed. If you want to preserve the returned value, you'll need to assign it to a variable.

## 6.4.2 NON-STANDARD EVALUATION

You may have noticed that in all of the previous `dplyr` verbs I provided the arguments (frequently column names such as `mpg`) without explicitly stating that I meant “the data from the `mpg` column of the `mtcars` dataset.” This may seem a little like magic, and that fact should make you slightly wary (but still impressed).

Don't let the name *non-standard evaluation* worry you — it just means that something special is going on behind the scenes to allow this to work. Internally, these functions check the local (defined within the function call) *scope* (discussed in chapter 4) to see if there's a column with that name, and if there is it uses that data as the values in the calculation. This makes life a *lot* easier when you're working in the Console or evaluating line-by-line in a script, because instead of typing things like

```
head(mutate(mtcars, hp_per_cyl = mtcars$hp /
mtcars$cyl)) ①
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
hp_per_cyl
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4     4
18.33333
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4     4
18.33333
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4     1
23.25000
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3     1
18.33333
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3     2
21.87500
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3     1
17.50000
```

①

**Uses an explicit reference to the data**

you can have some confidence that R will know to look in the `mtcars` data for these columns, so you can write

```
head(mutate(mtcars, hp_per_cyl = hp / cyl)) ①
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
hp_per_cyl
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4     4
18.33333
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4     4
18.33333
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4     1
23.25000
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3     1
18.33333
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3     2
21.87500
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3     1
17.50000
```

①

**Uses non-standard evaluation**

This makes it much easier to recognize what the calculation is doing. The danger comes about if you want to change those

columns programmatically, since you can't rely on this feature outside of the `dplyr` function. You can't write something like

```
column_to_divide <- cyl  
#> Error: object 'cyl' not found
```

(for the case where you might want to update that value later) because there is no variable `cyl` in the namespace. Instead, store the column name as a character:

```
column_to_divide <- "cyl"
```

You'd like to be able to write the following, but `mutate()` thinks you're trying to divide a numeric column by a string, so this fails:

```
head(mutate(mtcars, ratio = mpg / column_to_divide))
```

```
Evaluation error: non-numeric argument to binary operator.
```

Instead, tell R to *unquote* this back into a *symbol* using the `rlang` package. The gritty details of this are fairly advanced and difficult to navigate, leading to somewhat more difficult-to-read code, but it does work:

```
# install.packages("rlang")
library(rlang)
head(mutate(mtcars, ratio = mpg /
  !!sym(column_to_divide))) ①
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
ratio
#> 1 21.0   6   160 110 3.90 2.620 16.46   0   1     4     4
3.500000
#> 2 21.0   6   160 110 3.90 2.875 17.02   0   1     4     4
3.500000
#> 3 22.8   4   108  93 3.85 2.320 18.61   1   1     4     1
5.700000
#> 4 21.4   6   258 110 3.08 3.215 19.44   1   0     3     1
3.566667
#> 5 18.7   8   360 175 3.15 3.440 17.02   0   0     3     2
2.337500
#> 6 18.1   6   225 105 2.76 3.460 20.22   1   0     3     1
3.016667
```

①

**sym** converts the string into a symbol, whereas **!!** does the unquoting to extract the stored value from it.

You'll use the normal semi-magical recognition of arguments feature, but it would be wise to remember that with this come inherent limitations on where variables will be assumed to be found. Given a variable `carb`, take care about where R will look for it:

```
carb <- 1000
head(mutate(mtcars, double_carb = carb * 2))
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb
double_carb
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4     4
8
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4     4
8
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4     1
2
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3     1
2
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3     2
4
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3     1
2
```

**TIP** Name variables carefully.

To override this feature, you can be more explicit about where `dplyr` should look for certain variables (in the data you provide or in the environment?). To use the column `carb` in the preceding example (the default behavior) explicitly, refer to it as

`.data$carb`:

```
head(mutate(mtcars, double_carb = .data$carb * 2))
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb
double_carb
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4     4
8
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4     4
8
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4     1
```

```

2
#> 4 21.4    6   258 110 3.08 3.215 19.44   1   0     3   1
2
#> 5 18.7    8   360 175 3.15 3.440 17.02   0   0     3   2
4
#> 6 18.1    6   225 105 2.76 3.460 20.22   1   0     3   1
2

```

But to use the `carb` variable you also defined, you can refer to it as `.env$carb`:

```

head(mutate(mtcars, double_thousand = .env$carb * 2))
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
double_thousand
#> 1 21.0    6   160 110 3.90 2.620 16.46   0   1     4   4
2000
#> 2 21.0    6   160 110 3.90 2.875 17.02   0   1     4   4
2000
#> 3 22.8    4   108  93 3.85 2.320 18.61   1   1     4   1
2000
#> 4 21.4    6   258 110 3.08 3.215 19.44   1   0     3   1
2000
#> 5 18.7    8   360 175 3.15 3.440 17.02   0   0     3   2
2000
#> 6 18.1    6   225 105 2.76 3.460 20.22   1   0     3   1
2000

```

## Backticks and quotes to the rescue again

You may already be wondering how this deals with the special variables since you're using bare variable names that *may* in theory include spaces. If a column name has any irregular characters (spaces, punctuation, or start with a number or dot-number), you can still surround them with backticks or quotes, and `dplyr` will treat it like a bare name:

```

d <- data.frame(
  x = 21,
  `y variable` = 22,
  `.2b` = 23,
  check.names = FALSE
)

```

```
select(d, `y variable`, `.2b`)
#>   y variable .2b
#> 1      22 23
```

## 6.4.3 PIPES

The fact that the `.data` argument is *always* the first argument allows one further extension that, although it seems like a heavy departure from the way R has been written up to now, makes reading and writing R code a great deal easier on the user. The *pipe* operator `%>%` (a function, behind the scenes) takes whatever is on its direct left and inputs that to the first argument of whatever is on its direct right (another function, either written as a function *name* (`foo`) or a function *call* (`foo()`)). This looks like

```
x %>% y()
```

and can be read as “take `x` and then apply the function `y()`.” You can think of it as equivalent to writing

```
y(x)
```

since `x` has now been provided as the first (in this case, only) argument to `y()`. If the function on the right takes more arguments, these are provided as if the first were already there.

This works for any type of data, so you can do simple things, too, such as this:

```
c(100, 10, 1000) %>% ①
  log(base = 10) %>% ②
    mean() ③
#> [1] 2
```

①

**Given a vector of values...**

②

**...calculates the (vectorized) base-10 `log()` of these values, at this point equivalent to `log(c(100, 10, 1000), base = 10) = c(2, 1, 3)`...**

(3)

...and calculates the (vectorized) `mean()` of these values, at this point equivalent to `mean(log(c(100, 10, 1000), base = 10)) = mean(c(2, 1, 3)) = 2.`

That doesn't seem like a big deal until you have lots of functions all acting on top of each other, passing complicated objects into each other, as in the following:

```
summarize(
  group_by(
    select(
      filter(mtcars, cyl > 4),
      mpg, hp, am
    ),
    am
  ),
  mileage = mean(mpg)
)
#> # A tibble: 2 x 2
#>   am mileage
#>   <dbl>   <dbl>
#> 1 0     16.1
#> 2 1.00  18.5
```

(1)

**Summarizes by calculating the mileage as the mean of the mpg column**

(2)

**Groups by categories in the am column**

(3)

**Select columns mpg, hp, am**

(4)

**Filters mtcars to records with cyl greater than 4**

That calculates the mileage for automatic (`am == 1`) and manual (`am == 0`) transmission vehicles in the `mtcars` dataset that have more than 4 cylinders (`cyl > 4`). It's difficult to unravel, though. The `summarize` command is visually separated from the argument where the calculation is requested; the grouping by `am` sits out on its own away from the command itself. This is with the code written with a generous application of line breaks, too — more often, the calculation would be written on a single line

```
summarize(group_by(select(filter(mtcars, cyl > 4), mpg, hp,
am), am), mileage = mean(mpg))
#> # A tibble: 2 x 2
#>   am   mileage
#>   <dbl>    <dbl>
#> 1 0      16.1
#> 2 1.00   18.5
```

which is nearly indecipherable.

This “from the inside out” pattern of writing code is how functions are “supposed” to behave; the arguments are all in their correct places, but the *composition* (writing `f(g(x))`) makes it hard for *us* to read. R has no problem whatsoever, but why should we make it easiest for the computer when we’re the ones who will get angry when we can’t read it?

One solution is to split out the component function calls into more variables, such as one for

```
mtcars_filtered <- filter(mtcars, cyl > 4)
```

and so on. But this leads to the creation of a lot of variables that you only need for the next step, and naming things is hard.

The (vastly superior) alternative is the pipe: `%>%`. Instead of writing a function from the innermost data and adding functions around it, you can begin with the data, apply a function to it, then apply a function to whatever that produces, and so on, because the first argument to all of these functions is the data (what was just calculated). This becomes much easier to read, not least because the arguments to the functions stay within their parentheses. I hope you’ll agree that the following is much

clearer:

```
mtcars %>%
  filter(cyl > 4) %>%
  select(mpg, hp, am) %>%
  group_by(am) %>%
  summarize(mileage = mean(mpg))
#> # A tibble: 2 x 2
#>   am   mileage
#>   <dbl>    <dbl>
#> 1 0        16.1
#> 2 1.00    18.5
```

Yes, it produces the same output. You can see, line by line, what the steps in the processing are and the flow of the data through them. The `mtcars` data was inserted into the first argument of `filter`, that function was evaluated, that result was inserted into the first argument of `select`, and so on. This makes writing long chains of functions much simpler and easier to read when you need to figure out what you were doing. It also helps prevent mixing up arguments and data since you can clearly see what is happening in each step.

It may help build your mental model of what's happening here if you use a temporary variable just this once. If you store `mtcars` as the variable `.` (just a period, which is indeed valid, though it won't appear in the Environment pane since it begins with a period)

```
. <- mtcars
```

and then use this explicitly as the first argument to `filter()`, storing that result back into `.`

```
. <- filter(., cyl > 4)
```

and then explicitly use this as the first argument to `select()`, again storing that back into `.`

```
. <- select(., mpg, hp, am)
```

and so on, then you see that the `%>%` operator merely does this substitution for you. In fact, that's exactly what it does, and you

can even use the `.` value in functions to refer to the left-hand side of `%>%`, such as filtering to every fifth row:

```
mtcars %>%
  (seq_len(nrow(.)) %% 5 == 0)
#>   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> 1 18.7   8 360.0 175 3.15 3.440 17.02 0 0 3 2
#> 2 19.2   6 167.6 123 3.92 3.440 18.30 1 0 4 4
#> 3 10.4   8 472.0 205 2.93 5.250 17.98 0 0 3 4
#> 4 33.9   4  71.1  65 4.22 1.835 19.90 1 1 4 1
#> 5 19.2   8 400.0 175 3.08 3.845 17.05 0 0 3 2
#> 6 19.7   6 145.0 175 3.62 2.770 15.50 0 1 5 6
```

`nrow(.)` returns the number of rows of `..`, which in this case refers to `mtcars`. `seq_len(x)` creates the sequence `1, ..., x`. And `x %% y` calculates the remainder of dividing `x` by `y` as many times as possible, also known as *modulo*, for which the result will be `0` if `x` is an exact multiple of `y`.

There are many other advanced features of the pipe that we won't go into here, but support for this functionality is steadily growing among developers writing new packages. The pipe is available automatically if you load the `dplyr` package, or many other packages from the tidyverse. But its origin is in another package: `magrittr`, named as an allusion to Magritte's "This is not a pipe" painting, shown in figure 6.4.



**Figure 6.4** Magritte's “La Trahison des Images” (“The Treachery of Images”). The translation is, “This is not a pipe.”

## 6.4.4 SUBSETTING DATA.FRAME THE HARD WAY

dplyr should be your go-to for working with data.frames and tibbles, but it's also good to know how to work with the base operators relevant to these. It's hopefully not too surprising that you perform extraction on data.frames in the same way you do for a matrix. The same constructions (and caveats) apply here too. If you have the data.frame

```
my_df <- data.frame(
  x = c(1, 2, 3, 4),
  y = c("a", "b"),
  z = c(2.1, 9.3, 7.6, 1.1),
  stringsAsFactors = FALSE
)
```

then you can extract particular rows with [a, ], where the blank entry after the comma indicates that you want all columns:

```
my_df[3, ]
#>   x y   z
#> 3 3 a 7.6
```

Row names can also be used here the same as for a matrix, though I'll repeat that I don't recommend using row names for storing data. But the mtcars data has this feature, with the rownames stored as strings:

```
mtcars["Porsche 914-2", ]
#>           mpg cyl  disp hp drat    wt  qsec vs am gear
carb
#> Porsche 914-2 26     4 120.3 91 4.43 2.14 16.7  0  1      5
2
```

You can extract particular columns with [ , a], where the blank entry before the comma indicates that you want all rows:

```
my_df[ , 3]
#> [1] 2.1 9.3 7.6 1.1
```

Note that the latter of these is a vector, despite extracting from a data.frame:

```
class(my_df[, 3])
#> [1] "numeric"
```

The same `drop = TRUE` issue with `[` in chapter 5's matrices discussion is in effect here, but only when you select a single column. Selecting a single row (every column of) still produces a `data.frame`:

```
class(my_df[3, ])
#> [1] "data.frame"
```

To prevent this, explicitly request that the `drop` (to a vector) operation be avoided:

```
my_df[, 3, drop = FALSE]
#>      z
#> 1 2.1
#> 2 9.3
#> 3 7.6
#> 4 1.1
```

This can make the expression look particularly odd if you extract one particular *row* and *do* want the result to be dropped to a simpler structure (in this case, a list that will preserve the fact that a single row extracted across columns of different types will contain different types):

```
my_df[3, , drop = TRUE]
#> $x
#> [1] 3
#>
#> $y
#> [1] "a"
#>
#> $z
#> [1] 7.6
```

Forgetting the additional comma results in a column being extracted (perhaps unexpectedly) and a warning:

```
my_df[3, drop = TRUE]
# Warning message:
# In `.[data.frame](my_df, 3, drop = TRUE)`:
#   'drop' argument will be ignored
```

```
#>      z
#> 1 2.1
#> 2 9.3
#> 3 7.6
#> 4 1.1
```

Extracting with [, [[, or \$, partial matching of names, returning NULL for missing columns, and related issues all apply to `data.frames`.

The dollar-sign extraction and replacement methods work just the same too — and are more robust to re-orderings of data than using positional indices. You can extract columns

```
my_df$x
#> [1] 1 2 3 4
```

or replace elements within them:

```
my_df$z[2] <- 99
my_df
#>      x y      z
#> 1 1 a 2.1
#> 2 2 b 99.0
#> 3 3 a 7.6
#> 4 4 b 1.1
```

And you can use names to achieve this in the same way:

```
my_df[1, "x"] <- 99
my_df
#>      x y      z
#> 1 99 a 2.1
#> 2 2 b 99.0
#> 3 3 a 7.6
#> 4 4 b 1.1
```

You can also add new columns:

```
my_df$new_column <- c("red", "blue", "green", "white")
my_df
#>      x y      z new_column
#> 1 99 a 2.1      red
#> 2 2 b 99.0     blue
#> 3 3 a 7.6     green
#> 4 4 b 1.1     white
```

I personally do still use the dollar-name syntax for extracting a single column from a `data.frame` when working in the Console or with data where I *know* a certain column will have a certain name, as it's easily the fastest way to do so. When programming inside a function, though, it's much safer to use the square-bracketed column-selection tools.

## 6.5 Replacing NA

One frequently used pattern is replacing `NA` values with something. Perhaps these are more meaningful to an analysis as `0` values or some character string. You can combine conditionally selecting with replacement to achieve this, but be careful.

Comparing your data to `NA` won't work, since a comparison with `NA` always results in `NA`. As you saw earlier, trying to subset a vector with `NA` produces a vector of `NA` values. Instead, you can use the `is.na()` function we looked at in section 3.4:

```
x <- c(7, NA, 9, NA)
is.na(x)
#> [1] FALSE  TRUE FALSE  TRUE
```

Because this produces a logical vector, you can use it to subset some data and then replace those `NA` values with something else. Given a `data.frame` with `NA` values

```
d_NA <- data.frame(x, y = 1:4)
d_NA
#>   x y
#> 1 7 1
#> 2 NA 2
#> 3 9 3
#> 4 NA 4
```

you can replace `NA` with `0` by selecting the rows for which `x` is `NA`, and the column `x`:

```
d_NA[is.na(d_NA$x), "x"] <- 0
d_NA
#>   x y
#> 1 7 1
```

```
#> 2 0 2
#> 3 9 3
#> 4 0 4
```

`is.na()` also has a `data.frame` *method* (discussed in chapter 5) so you can pass the whole `d_NA` object in, and this will return a logical `data.frame`:

```
d_NA <- data.frame(x, y = 1:4)
is.na(d_NA)
#>           x     y
#> [1,] FALSE FALSE
#> [2,] TRUE  FALSE
#> [3,] FALSE FALSE
#> [4,] TRUE  FALSE
```

That means you can use this condition for replacement:

```
d_NA <- data.frame(x, y = 1:4)
d_NA[is.na(d_NA)] <- 0
d_NA
#>   x y
#> 1 7 1
#> 2 0 2
#> 3 9 3
#> 4 0 4
```

There's a helper function to do the opposite (insert `NA` values into a vector). The function is `is.na<-`, which means you can use it in a somewhat odd way:

```
d_NA <- data.frame(x = 7:10, y = 1:4)
is.na(d_NA$x) <- c(1, 3) ①
d_NA
#>   x y
#> 1 NA 1
#> 2  8 2
#> 3 NA 3
#> 4 10 4
```

①

Specifies which values in the vector to set to NA

Because `is.na()` produces a logical vector, you can use the

inverse of this by prepending it with the negation operator `!`, so requesting the non-NA values looks like this:

```
y <- c(3, 4, NA, NA, 7)
y[!is.na(y)]
#> [1] 3 4 7
```

## 6.6 Selecting conditionally

All the methods of extracting from the structures we've discussed continue to work if, instead of specifying the rows, columns, or elements to extract, you provide a condition that determines those. Chapter 5's discussion of vectors lets you know that automatic recycling of vectors can make comparisons much simpler and more compact:

```
test_nums <- c(6, 5, 4, 5, 7, 4)
test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Rather than provide a selection of indices to `[` or `[[`, you can instead provide a logical vector of the same length as the row or column you are extracting from, with `TRUE` representing elements you want to extract, and `FALSE` representing those you don't:

```
goodBad <- c("good", "bad", "good", "good", "bad")
goodBad[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
#> [1] "good" "good" "good"
```

Automatic recycling will again take place if the logical vector you supply isn't long enough:

```
goodBad[c(TRUE, FALSE)]
#> [1] "good" "good" "bad"
```

The vector will be partially recycled if its length isn't a multiple of the object's length.

Because vectors themselves can be used to generate these logical vectors, as in

```
goodBad == "good"
#> [1] TRUE FALSE TRUE TRUE FALSE
```

you can avoid having to generate this yourself, and rely only on variable names:

```
goodBad[goodBad == "good"]
#> [1] "good" "good" "good"
```

With `data.frames`, it's more complicated, because you need to refer to the `data.frame` to reference columns. For example, to perform the test of whether the values of `cyl` in the `mtcars` dataset are equal to 6, you can use automatic recycling:

```
mtcars$cyl == 6
#> [1] TRUE TRUE FALSE TRUE FALSE TRUE FALSE FALSE
FALSE TRUE TRUE
#> [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE
#> [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
FALSE FALSE

# equivalently mtcars[["cyl"]] == 6
```

You could follow the pattern used in the earlier `goodBad` example and extract the elements of `mtcars$cyl` that meet this criteria, but that's not terribly useful. Instead, you can use this logical vector to extract only the *rows* for which the logical vector is `TRUE`, if you use the `[row, column]` syntax, remembering that leaving one of these options blank retrieves all rows or columns:

```
mtcars[mtcars$cyl == 6, ]
#>             mpg cyl  disp  hp drat    wt  qsec vs am
gear carb
#> Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1
4     4
#> Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1
4     4
#> Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0
3     1
#> Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0
3     1
#> Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0
4     4
#> Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0
4     4
#> Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1
5     6
```

This is the slice of the `mtcars` dataset for which the values in the `cyl` column are 6. The calculation that produces the logical vector can be complicated and involve other columns too. To restrict this further to only rows where `am == 1`, you could do this:

```
mtcars[mtcars$cyl == 6 & mtcars$am == 1, ]
#>          mpg cyl disp hp drat    wt  qsec vs am
gear carb
#> Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1
4      4
#> Mazda RX4 Wag 21.0     6 160 110 3.90 2.875 17.02  0  1
4      4
#> Ferrari Dino 19.7     6 145 175 3.62 2.770 15.50  0  1
5      6
```

The calculation that builds this logical vector doesn't need to involve the dataset you're slicing. It can be any calculation that results in a logical vector, such as this one:

```
mtcars[1:8 > 7, ]
#>          mpg cyl disp hp drat    wt  qsec
vs am gear carb
#> Merc 240D        24.4     4 146.7   62 3.69 3.190 20.00
1  0     4     2
#> Lincoln Continental 10.4     8 460.0 215 3.00 5.424 17.82
0  0     3     4
#> Camaro Z28        13.3     8 350.0 245 3.73 3.840 15.41
0  0     3     4
#> Volvo 142E        21.4     4 121.0 109 4.11 2.780 18.60
1  1     4     2
```

That selects every eighth row, because the first seven values are not greater than 7 and the last is (the resulting vector is recycled sufficiently):

```
1:8 > 7
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

Using this syntax to select columns works — but as I've said, using the positional index of something is fragile, so I don't recommend doing so. Nonetheless, you can extract every second column like this:

```
head(
```

```

mtcars[, c(FALSE, TRUE)]
)
#>          cyl  hp   wt vs gear
#> Mazda RX4      6 110 2.620  0   4
#> Mazda RX4 Wag  6 110 2.875  0   4
#> Datsun 710     4  93 2.320  1   4
#> Hornet 4 Drive 6 110 3.215  1   3
#> Hornet Sportabout 8 175 3.440  0   3
#> Valiant        6 105 3.460  1   3

```

To be a little more robust and select certain column names with a logical vector, use the fact that `colnames()` returns a vector. You can create a calculation that's as complicated as you want:

```

cn <- colnames(mtcars)
head(
  mtcars[, startsWith(cn, "c") | endsWith(cn, "p")]
)
#>          cyl disp  hp carb
#> Mazda RX4      6 160 110    4
#> Mazda RX4 Wag  6 160 110    4
#> Datsun 710     4 108  93    1
#> Hornet 4 Drive 6 258 110    1
#> Hornet Sportabout 8 360 175    2
#> Valiant        6 225 105    1

```

Here I've used the (base) functions `startsWith()` and `endsWith()` that you saw in the discussion of text matching in section 6.1.1 to test whether the `colnames()` of `mtcars` starts/ends with a particular letter. The `|` operator *or* means that the result will be true if either match is found.

If, instead of a full logical vector, you just want to know *which* values satisfy a condition, you can pass the vector to the `which()` function to produce just the indices of the values that are `TRUE`. For example, the previous example searched the column names of `mtcars` for names starting with `c`:

```

startsWith(colnames(mtcars), "c")
#> [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE  TRUE

```

You can turn that into the column *numbers* that match this criteria:

```
which(startsWith(colnames(mtcars), "c"))
#> [1] 2 11
```

Subsetting with [ can take either a full logical vector or just the indices to keep, so either of these can be used to extract the column names that start with c:

```
colnames(mtcars)[which(startsWith(colnames(mtcars), "c"))]
#> [1] "cyl"   "carb"
```

All that said, the dplyr syntax makes for much cleaner code and achieves the same result (up to inclusion of row names). To select columns with names that start with "c", you can use the following:

```
mtcars %>%
  select(starts_with("c")) %>%
  head()
#>          cyl carb
#> Mazda RX4     6   4
#> Mazda RX4 Wag 6   4
#> Datsun 710    4   1
#> Hornet 4 Drive 6   1
#> Hornet Sportabout 8   2
#> Valiant        6   1
```

Or to select columns with names either starting with "c" or ending with "p", use a regular expression and the matches() helper:

```
mtcars %>%
  select(matches("^c|p$")) %>% ①
  head()
#>          cyl disp  hp carb
#> Mazda RX4     6 160 110   4
#> Mazda RX4 Wag 6 160 110   4
#> Datsun 710    4 108  93   1
#> Hornet 4 Drive 6 258 110   1
#> Hornet Sportabout 8 360 175   2
#> Valiant        6 225 105   1
```

①

The caret ^ matches the start of a string, and the dollar \$ matches the end of a string.

## 6.7 Summarizing values

Once you have subsetted data, you'll likely want to use it to calculate some summary value(s). The simplest way to do that is the `summary()` function, which is very useful. It can work with almost any structure of data and provides an overview summary that is specific to the structure. For a numeric vector, it shows the following five-number summary:

```
summary(3:9)
#>   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
#>   3.0    4.5    6.0    6.0    7.5    9.0
```

This details the range of values (minimum and maximum), the mean and median, and the first and third quartiles.<sup>4</sup> This shows how the data is distributed. The mean won't always be equal to the median, in particular if the data is skewed:

---

<sup>4</sup> If the data was sorted and divided into four equally sized groups, these would be values at the first and third internal boundaries. The second boundary defines the median.

```
set.seed(1)
summary(rlnorm(10))
#>   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
#>  0.4336  0.5851  1.2959  1.5166  1.7409  4.9297
```

When the vector is characters, `summary()` just tells about the characteristics of the vector:

```
summary(letters)
#>   Length   Class    Mode
#>      26 character character
```

It's a little more helpful when the data is `factor`, in which case `summary()` produces a count of the values in each `factor` level:

```
summary(iris$Species)
#>   setosa versicolor virginica
#>      50        50        50
```

For a matrix or a `data.frame`, the `summary()` function works over columns and details each of these. For numeric columns, it produces the five-number summary as before. For factor columns, it details the counts of each value (if there are only a few):

```
summary(iris[, 3:5])
#>   Petal.Length   Petal.Width      Species
#>   Min.   :1.000   Min.   :0.100   setosa    :50
#>   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
#>   Median  :4.350   Median  :1.300   virginica :50
#>   Mean    :3.758   Mean    :1.199
#>   3rd Qu.:5.100   3rd Qu.:1.800
#>   Max.    :6.900   Max.    :2.500
```

This is also useful when values are missing from data, as they will be counted below each column's summary. Say you have a `data.frame` with missing values, an extract of the `mtcars` data:

```
mtcars_NA <- mtcars[, 1:4]

mtcars_NA[1, 4] <- NA ①
mtcars_NA[2, 3] <- NA ①
mtcars_NA[3, 4] <- NA ①
mtcars_NA[4, 1] <- NA ①
mtcars_NA[4, 3] <- NA ①

head(mtcars_NA)
#>          mpg cyl disp  hp
#> Mazda RX4     21.0   6 160 NA
#> Mazda RX4 Wag 21.0   6  NA 110
#> Datsun 710    22.8   4 108 NA
#> Hornet 4 Drive NA    6  NA 110
#> Hornet Sportabout 18.7   8 360 175
#> Valiant       18.1   6 225 105
```

①

Replaces individual values with NA

The first line involves extracting the first four columns of the `mtcars` data you've been working with, saved to another object for clarity. Even if you modified the `mtcars` object, the original would still be available as `datasets::mtcars`.

The `summary()` of this will let you know how many values are NA in each column:

```
summary(mtcars_NA)
#>      mpg          cyl          disp
hp
#>  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.
: 52.0
#>  1st Qu.:15.35   1st Qu.:4.000   1st Qu.:120.5   1st
Qu.: 99.0
#>  Median :19.20   Median :6.000   Median :196.3   Median
:136.5
#>  Mean    :20.05   Mean    :6.188   Mean    :232.2   Mean
:149.7
#>  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:342.0   3rd
Qu.:180.0
#>  Max.    :33.90   Max.    :8.000   Max.    :472.0   Max.
:335.0
#>  NA's     :1                   NA's     :2                   NA's
:2
```

`summary()` also works with more complex structures — we'll return to these later.

To distill the data down, you need another function. The `aggregate()` function from the built-in `stats` package is one way to achieve this for `data.frames`. It takes three main arguments:

- `x` — The `data.frame` to aggregate
- `by` — A list of grouping elements
- `FUN` — The function to be used to compute the summary

In practice, grouping the `iris` dataset (not including the fifth column — the `Species`) by the different values of `Species` to calculate the mean of each column looks like the following:

```
aggregate(x = iris[, -5], by = list(Species =
iris$Species), mean)
#>      Species Sepal.Length Sepal.Width Petal.Length
Petal.Width
#> 1    setosa      5.006      3.428      1.462
0.246
#> 2 versicolor   5.936      2.770      4.260
1.326
```

```
#> 3 virginica      6.588      2.974      5.552
2.026
```

## Formula interface

There is another way to use aggregate: with a formula. This looks a bit like an equation with a value on the left, a tilde (~), and a value on the right, like this:

```
y ~ x
```

The way to read that is “y as a function of x.” You can use this to define a grouping instead of the list:

```
aggregate(formula = . ~ Species, data = iris, FUN =
mean)
#>           Species Sepal.Length Sepal.Width Petal.Length
Petal.Width
#> 1       setosa      5.006      3.428      1.462
0.246
#> 2 versicolor     5.936      2.770      4.260
1.326
#> 3 virginica      6.588      2.974      5.552
2.026
```

Keep in mind that, for whatever reason, the arguments are ordered differently now, so either make sure to use their names or respect the relevant ordering.

This formula has a dot (.) that stands for “all columns” on the left, and Species (note: not `iris$Species`) on the right, so the grouping is defined to be “all columns as a function of Species.”

Internally, the components of a formula are stored as a list, and the function knows to look for the data in the relevant structure, in the same way you saw in section 6.4.2.

I can’t say I’m a fan of this function; `dplyr`’s grouped `summarize()` is a much cleaner option.

## 6.8 A worked example: Excel vs. R

You're finally able to begin working with real data in a meaningful way. It's taken a while, but what you've seen along the way will all come in handy right now.

Let's take the `mtcars` dataset and ask some questions that you might have previously answered using a spreadsheet program. First, what's the average mileage (miles per gallon, `mpg`) of the cars in this data? If you had this data in Excel, you might calculate `=AVERAGE(B2:B33)` if `mpg` was the second column. Can you use your new R skills to answer this same question? Create and save a new script file, and write some code that produces this result. The `mtcars` dataset is already available in your session.

There are many ways to do this in R. A straightforward way would be to extract the relevant column from the data as a vector and calculate the `mean()` from that:

```
mean(mtcars$mpg)
#> [1] 20.09062
```

You could alternatively use the `dplyr` package (which is overkill at this point, but you'll quickly expand to wanting this):

```
library(dplyr)
summarize(mtcars, mileage = mean(mpg))
#>   mileage
#> 1 20.09062
```

Note also that this produces a `data.frame` with just a single row and column. You could convert that to a number by passing it to `as.numeric()`:

```
as.numeric(summarize(mtcars, mileage = mean(mpg)))
#> [1] 20.09062
```

Simple enough.

You could use the pipe here, too:

```
library(dplyr)
```

```
mtcars %>%
  summarize(mileage = mean(mpg)) %>%
  as.numeric()
#> [1] 20.09062
```

The answer using R should match the one you produce using Excel.

Now let's ask another question: what is the mileage for the automatic transmission cars (`am == 0`)? In Excel, you might use a filter to select only those records, and at this point you may notice where the approaches differ. By applying the filter, you have modified the state of your data. Not in a permanent way of course (yet), but you can no longer compare the new average to what you calculated for the full dataset without undoing the filter.

There are typically two ways around this: creating a copy of the data so that the two calculations can coexist; or being more explicit in the calculation to perform the filtering in place, such as with `=AVERAGEIF(I2:I33, "=0", B2:B33)` (average the values in rows 2 to 33 of the second column if the corresponding value in the ninth column is 0). The first of these options is unsatisfactory because there is no guarantee that an update to the original data will propagate to the copy. The second option, I would argue, is a programming solution, one that is better suited to a more descriptive programming language. In R, you can add this calculation without modifying or saving unnecessary copies of the data as so:

```
mtcars %>%
  filter(am == 0) %>%
  summarize(mileage = mean(mpg)) %>%
  as.numeric()
#> [1] 17.14737
```

You can swap back and forth between these two results by commenting out the `filter()` step:

```
mtcars %>%
#  filter(am == 0) %>%      ①
  summarize(mileage = mean(mpg)) %>%
```

```
as.numeric()
#> [1] 20.09062
```

①

**Note that this line won't be evaluated.**

You can be assured that you're comparing what you intended to.

If you have both in your script file, any change to the input data will be reflected in your calculations since you refer to the data with the same variable name (`mtcars` in the example, but this would be whatever you name your data). You can ask and answer as many complex questions as you like, storing the results you want to keep in new variables; and by writing the code in a script, you preserve the steps that produce those values. Go back to the last Excel file you were working on, and ask yourself, “What have I calculated?” Is the filter still active? Is it clear that whatever calculation you’ve entered into a cell currently represents the answer you were trying to produce? Could you give this to someone and expect them to extract the same answer to your question?

Spreadsheet programs are excellent for displaying tabular data sets and are capable of generating answers to questions, but they are rarely used in a manner that is reproducible and robust. By scripting your analysis and working with unchanging source data in a descriptive programming language,<sup>5</sup> you create a reproducible flow from question to answer.

---

<sup>5</sup> It is worth noting at this point that there is a way to achieve this in Excel; VBA is a programming language that interfaces with Excel.

## 6.9 Try it yourself

The `mtcars` data set is available in your session. Try to answer the following questions using both the base subsetting functions as well as `dplyr`'s verbs.

Using `mtcars`:

- What is the maximum (`max()`) weight of the automatic (`am == 1`) vehicles?
- What is the minimum (`min()`) weight of the manual (`am == 0`) vehicles?
- What is the mean displacement (`disp`) of six-cylinder (`cyl`) vehicles?
- What is the mean displacement (`disp`) of four-cylinder (`cyl`) vehicles?
- What is the mean displacement of vehicles grouped by number of cylinders?

Don't peek at the following solutions, but feel free to go back through this chapter to make sure you're comfortable with how to do these.

### **6.9.1 SOLUTIONS — NO PEEKING**

One way to achieve these results in each of the base and `dplyr` styles would be as follows:

- What is the maximum (`max()`) weight of the automatic (`am == 1`) vehicles?

```
max(mtcars[mtcars$am == 1, "wt"])
#> [1] 3.57
```

```
mtcars %>%
  filter(am == 1) %>%
  summarize(max(wt)) %>%
  as.numeric()
#> [1] 3.57
```

- What is the minimum (`min()`) weight of the manual (`am == 0`) vehicles?

```
min(mtcars[mtcars$am == 0, "wt"])
#> [1] 2.465
```

```
mtcars %>%
  filter(am == 0) %>%
```

```
summarize(min(wt)) %>%
  as.numeric()
#> [1] 2.465
```

- What is the mean displacement (`disp`) of six-cylinder (`cyl`) vehicles?

```
mean(mtcars[mtcars$cyl == 6, "disp"])
#> [1] 183.3143
```

```
mtcars %>%
  filter(cyl == 6) %>%
  summarize(mean(disp)) %>%
  as.numeric()
#> [1] 183.3143
```

- What is the mean displacement (`disp`) of four-cylinder (`cyl`) vehicles?

```
mean(mtcars[mtcars$cyl == 4, "disp"])
#> [1] 105.1364
```

```
mtcars %>%
  filter(cyl == 4) %>%
  summarize(mean(disp)) %>%
  as.numeric()
#> [1] 105.1364
```

- What is the mean displacement of vehicles grouped by number of cylinders?

```
aggregate(
  x = mtcars[, "disp", drop = FALSE],
  by = list(cyl = mtcars$cyl),
  FUN = mean
)
#>   cyl      disp
#> 1   4 105.1364
#> 2   6 183.3143
#> 3   8 353.1000
```

This one was a bit of a hassle since you still need `x` to be a `data.frame` in order to retain the column name `disp` in the result, and by default R tries to drop this single-column `data.frame` to a vector:

```
mtcars %>%
  group_by(cyl) %>%
  summarize(mean_disp = mean(disp))
#> # A tibble: 3 × 2
#>   cyl    mean_disp
#>   <dbl>      <dbl>
#> 1     4.00      105.
#> 2     6.00      183.
#> 3     8.00      353.
```

The `print()` method for `tibbles` tries to format numbers nicely and in this case shows only so much precision. The data has the right values, though.

## ***Terminology***

- *Regular expression* — A powerful way of specifying abstract components within text to which a match can be made.
- *Closure* — Proper name for a function, enclosing both an expression and an environment.
- *dplyrverbs* — Action words (*run, eat, play*), are commonly used to name functions by what action they perform (`mutate`, `select`, `filter`).
- *Atomic vector* — R’s basic vector types (for example, logical, numeric, character) are the simplest types of vectors. Technically, lists are also vectors, but they’re not *atomic*, they’re *recursive*.

## ***Summary***

- You can subset text most easily with functions from the `stringr` package.
- Regular expressions make matching text patterns easier.
- `[` and `[[` can be used to subset most structures, but with many corner cases.
- `[[` selects only a single element.

- Subsets can be extracted using `$` for named subsets, such as a column of a `data.frame`.
- An assignment on a subset replaces the subset, for example `vec[2] <- 99`.
- The `dplyr` package makes working with `data.frames` easier.
- Non-standard evaluation changes the way that columns are identified within many functions.
- The pipe `%>%` makes chaining operations together easier to read (and write).
- Subsets can be created by conditional selections using logical vectors.
- Regular expressions (even those not involving special selectors) are case sensitive.
- The pattern substitution function `sub()` only performs an operation once; to perform it on every match, use `gsub()`.
- Special characters (such as `\`) need to be escaped to be used in regular expressions.
- Dollar-name selection will partially match to names by default, but `[[` won't.
- When a `data.frame` subset can be converted to a vector, it will be by default (use `drop = FALSE` to prevent this).
- The `tibble` class `tbl_df` is also a `data.frame`, so functions that take a `data.frame` should also be able to take a `tibble`.
- The first argument of `dplyr` verbs is `.data`, but this isn't always true for other functions in other packages, so the pipe (`%>%`) won't work out of the box with every function.

# 7

## *Doing things with lots of data*

---

### This chapter covers

- How to read data into R from different sources
- How to inspect your data once it's in R
- How to write R data to a file

Working with the example data sets has been useful; they're not overly complex, yet they have features that help demonstrate the tools you've been learning to use. But it's not *your* data, and that's what you really want to use; so let's explore how to get your data into your workspace, ready to be manipulated/interrogated, and then store it for safe keeping once you're done.

### **7.1 Tidy data principles**

If you've ever received data from someone else, you've likely encountered the issue of them not sharing your mental model of how that data *should* look. Even something as simple as a table of values can be structured in many different ways, depending on who created it.

If you have, say, the heights, weights, and ages of a group of people,<sup>1</sup> you might find these stored as a table with repeated labels (one for each person) with a descriptive label for the thing

being measured (the *variable*) and a *value* corresponding to that variable. This could be entered as follows:

---

<sup>1</sup> See the Wikipedia article at <http://mng.bz/Weoo>. The data is partial census data for the Dobe area !Kung San, compiled from interviews conducted by Nancy Howell in the late 1960s, available from <http://mng.bz/JAKV>.

---

```
long_df <- data.frame(
  id = c(
    "person1", "person2", "person3", "person1", "person2",
    "person3", "person1", "person2", "person3", "person1",
    "person2", "person3"
  ),
  variable = c(
    "height", "height", "height", "weight", "weight",
    "weight", "age", "age", "age", "male", "male", "male"
  ),
  value = c(151.8, 139.7, 136.5, 47.8, 36.5,
            31.9, 63, 63, 65, 1, 0, 0),
  stringsAsFactors = FALSE
)
long_df
#>           id variable value
#> 1  person1   height 151.8
#> 2  person2   height 139.7
#> 3  person3   height 136.5
#> 4  person1   weight  47.8
#> 5  person2   weight  36.5
#> 6  person3   weight  31.9
#> 7  person1      age  63.0
#> 8  person2      age  63.0
#> 9  person3      age  65.0
#> 10 person1     male   1.0
#> 11 person2     male   0.0
#> 12 person3     male   0.0
```

This *long format* isn't uncommon and can be useful for reading off pairs of variables and values or subsetting to certain variables or values. A more usefully structured format, though, is to use *tidy* data principles, under which you store data with one row per subject and one column per observed quantity. In the preceding example, this means one row per person, and one row per thing being measured (height, weight, age, male/female).

This compacts the table down to a *wide format*, where multiple values can be compared across a row:

```
wide_df <- data.frame(
  id = c("person1", "person2", "person3"),
  height = c(152, 140, 137),
  weight = c(47.8, 36.5, 31.9),
  age = c(63, 63, 65),
  male = c(1, 0, 0),
  stringsAsFactors = FALSE
)

#>      id height weight age male
#> 1 person1    152    47.8  63    1
#> 2 person2    140    36.5  63    0
#> 3 person3    137    31.9  65    0
```

Subsetting can still be performed, but you now have the benefit of being able to specify by *which* variables to subset values. This is the structure to aim to have data in, either before or after receiving it. It will also come in handy later when certain package functions expect data in this format. A group of widely used packages adopts and expects this tidy principle when using the majority of their functions, and these have collectively come to be known as the *tidyverse*. They are primarily written by `dplyr`'s author Hadley Wickham, but the tidyverse has expanded to include a wide variety of packages by many authors and contributors.

Don't worry if your data isn't currently in this format — we'll get to how to go between these two options soon enough. For now, just know the difference and try to classify your data as one format or the other (wide or long).

### **7.1.1 THE WORKING DIRECTORY**

When you start an R session and assign values to variables in the Workspace, you're doing so without any specific mention of folders on the computer. The R Workspace exists only within computer memory, but the R session nevertheless has a connection to the directory structure via the *working directory*. This is where, by default, files will be read from and to, and it's

also where you'll find any temporary or ancillary files that R or RStudio creates.

By default, this is the user's home directory (My Documents on Windows, /home/<username> on Linux, /Users/<username> on Mac, with the cross-platform shortcut ~/). You can ask R what directory is currently listed as the working directory with the `getwd()` (get working directory) function, which takes no arguments:

```
getwd() ①
#> [1] "/home/user"
```

①

**The result of this will be specific to your computer.**

Any files to reference (such as reading from or saving to) within this working directory can be called by name:

```
xyz_data <- read.csv("xyzdata.csv") ①
```

①

**In this case, the file xyzdata.csv is a file in the current working directory: for example, /home/user/xyzdata.csv.**

RStudio will even peek inside subfolders and help to autocomplete their locations for you. You can still reference files with their full path name, but that requires you to give the *absolute* reference (starting from / or ~/ or c:/ and so on.)

If you ever need to share R code with someone (or even with yourself on another computer), then whoever is looking at it will be fairly disappointed to see

```
important_data <-
read.csv("D:/Path_to_my_files/important_data.csv")
```

because they won't be able to do anything with the code unless they either change all of the file references or have their own D:/Path\_to\_my\_files/ directory.

If you are working within an RStudio project (as discussed in section 1.4 — you may want to review that material), then the `here` package will help you manage files relative to that project’s root directory:

```
# install.packages("here")
library(here)
#> here() starts at /home/user/myProject
```

This cleverly figures out where your project is stored and allows you to reference files relative to that directory. You don’t have to store the full path to your data/files — you can store everything under a project directory, so if you share the project with someone (for example, if you host it on GitHub), they too will have all the correct file references because `here()` will always refer to that project’s root directory.

If the person you share code with has a copy of the project files with the following lines, they can rest assured that they will know where to look for `important_data.csv`, regardless of where they’ve saved their copy of your project:

```
library(here)
important_data <- read.csv(
  here("subdirectory_of_project",
  "important_data.csv") ①
)
```

①

**Subdirectories are provided as individual arguments, but the equivalent is "subdirectory\_of\_project/important\_data.csv".**

To change this working directory manually — say, to a specific folder to store all the files for an analysis — you can do that with the `setwd()` function (set working directory), which takes a character string describing the location of the folder to set as the working directory:

```
setwd("/path/to/my/analysis/folder/")
```

RStudio will help create this string by autocompleting (press Tab

partway through typing a valid folder path).

**CAUTION** Depending on which operating system you use, you'll need to take care with the folder path. On Windows systems, the separator between folders is a backslash (\); but as you saw in section 2.1.2, R treats this as a special character escape code, so you need to use *two* of them (escape the backslash with another backslash), as in C:\\Users\\username. Alternatively, turning the slashes around, as in C:/Users/username, also works.

There is also a more visual way to set the working directory: in RStudio, clicking Session > Set Working Directory provides several options, including setting the working directory to the current file's location or any other folder you choose.

**CAUTION** Be aware that when you start a new session, the working directory may have reverted to a default. Also be aware that other people won't necessarily have access to your computer when you share a script with them, so setting the working directory and referencing data relative to that will mean they can't reproduce your results. Instead, a better solution is to work with RStudio Projects (covered in section 1.4.1), in which case all directory and file references can be relative to the project root, made even easier with the here package.

## 7.1.2 STORED DATA FORMATS

If you've come from working with spreadsheets, then you're likely most familiar with having data in a file format relevant to the program you use, such as .xls or .xlsx for Excel, .ods for OpenOffice or LibreOffice, and .numbers for Numbers. These formats store more than just data — they also store the formatting you use, where any plots were located, the formulas,

colors, and so on. This used to be a big hurdle for getting data into R, but things have become a lot easier in the last few years, and this format *can* (under various conditions, especially when the data is simply structured) be processed by R. These formats have their own issues, though. For one thing, they don't interact well with version control systems (VCS — see chapter 1 for more).

Better suited to version control are plain-text files. At their simplest, these are just lines of text (letters, numbers, and symbols) in a file with some specific structure to them. A VCS (such as git) can compare two versions of the file and show what the differences are (if any). One of the most common plain-text data file structures is .csv (comma-separated values). Data in this format is stored as values separated by commas (shocking, I know). These files can be opened in a text editor, in which case you would see something like this:

```
id,height,weight,age,male
person1,152,47.8,63,1
person2,140,36.5,63,0
person3,137,31.9,65,0
```

Note that there are no spaces on either side of the commas, just the data values with a comma between them. Spaces are allowed in this format,<sup>2</sup> though great care needs to be taken if any of the text values themselves contain a comma. That can happen if the data happens to be natural language text, such as lines from a book or content from the web. Another likely source of commas is decimals. Some countries use a period (.) and some use a comma (,) for decimals.

---

<sup>2</sup> There is a proper definition of the CSV format (<https://tools.ietf.org/html/rfc4180>), but this in no way ensures that it is being followed.

---

A similar problem arises for large numeric values written with a comma as a thousands separator. In this case, it's better to manage the data in a slightly different format.

A straightforward variation on this would be to use a different record delimiter, such as ;, tab, space, or |. Each of these can be dealt with in R, but the same constraints will apply in any case:

- Values in fields (records) are on a line separated by a delimiter.
- The same number of fields (even if they're blank) are on a line.
- Lines end with a carriage return (line break).
- The delimiter can't be used *within* a record.
- Take care with escape characters (\).
- There's an optional first header line with the names of the fields.

When different delimiters are used, the format is sometimes referred to by that name, such as tab-separated values (TSV):

	<code>id</code>	<code>height</code>	<code>weight</code>	<code>age</code>	<code>male</code>
person1	152	47.8	63	1	
person2	140	36.5	63	0	
person3	137	31.9	65	0	

A more general format forgoes the tabular structure of these and places one record on each line in the file. This could be lines of a poem, names of items in stock, or recordings of a measurement made by a logger. There are fewer restrictions on what this data can contain, but you still need to take care with regard to escape characters.

Text files are the most common way to carefully store data, but they're far from the only way. R can handle almost *any* type of data in a file, be that images, GPS-logger files, shapefiles, audio, video, data files from other software packages, databases — just about anything you have.

**NOTE** There are limitations to this; the packages made to work with those formats may have been written long ago by people no longer maintaining them, they may not deal explicitly with the nuances of your file, and they may require special configuration of your system. This is hardly the fault of R and its packages, the

authors of which had no say in how those formats were constructed, and in many cases those formats are deliberately made difficult to read by a non-free software company.

Have a look at the file type of some of your data and identify which it is. Next, you'll work on pointing R to it and reading its contents into memory.

### 7.1.3 READING DATA INTO R

R has *many* different ways to get data from a stored file into memory, and in most cases there's a function best suited to the particular data structure. For general tabular data, the `read.table()` function has 25 configurable options to specify exactly how to read in data. The first of these, `file`, tells R where to look for the file, and its name (with extension). This is provided as a character string either with the full directory path, or relative to the working directory. The returned value of this function, if successful, is a `data.frame` representing the contents of the file, and by default this is printed to the Console unless you save it to a variable:

```
myTabularData <- read.table(file = "myFile.dat")
```

**NOTE** This function doesn't set up a *connection* to the contents of that file — it merely opens the file, reads the values into memory, and then closes the file. Any changes you make to either the file or the variable containing the data after this point won't be reflected in the other, unless you explicitly read/write again.

The remaining options are set to reasonable defaults but may not meet your specific needs. If you have CSV data, you would need to change the `sep = ""` (blank) option to `sep = ", "`. Thankfully, there are convenience functions for several common formats, `read.csv()` and `read.delim()`, which internally call `read.table()`

but have more specific defaults set.

If it's feasible and makes sense to do so (data in a single table), then converting a proprietary format (such as .xlsx) to CSV can make reading the data into R a lot simpler. But it's important to remember that once you make a copy of the data, the link back to the original is broken, so any changes you make in either won't be in sync.

These functions aren't limited to working with local files; the `read.table()` help documentation for the `file` argument notes that this can also be a URL (web link), so you can pull in data from a remote site:

```
Howell_data <- "http://mng.bz/JAKV"
head(①
  read.table(
    file = Howell_data, ②
    sep = ";", ③
    header = TRUE ④
  )
)
#>   height   weight age male
#> 1 151.765 47.82561 63   1
#> 2 139.700 36.48581 63   0
#> 3 136.525 31.86484 65   0
#> 4 156.845 53.04191 41   1
#> 5 145.415 41.27687 51   0
#> 6 163.830 62.99259 35   1
```

①

**Prints the first six rows to preview**

②

**Uses the string stored in the variable `Howell_data` as the file argument value**

③

**This file uses ; as a record separator.**

(4)

**The column names are provided on the first row, so use this option.**

This can be very useful for several reasons; you may want to store the offsite location of the data and reference it in a script to increase reproducibility (you won't need to share the data, or the source may be more trustworthy than an emailed data file). You may be working on data that is updated by someone else and hosted, and storing that data may not be necessary for analysis.

For example, if you were processing rainfall data provided by the local city's weather bureau, and wanted to produce a plot of the daily rainfall for the year, you might be satisfied to download the entire series of values each day instead of trying to only download a single day of values and maintain your own dataset (which transfers the responsibility of data assurance to us, and which may be more trouble in any case).

**CAUTION** It's quite common for spreadsheets to have column names with spaces or punctuation or that start with numbers — all the things that standard `data.frame` column names can't have (without special treatment). Recall from section 6.4 that spaces and punctuation will be converted to dots, whereas columns starting with a number (or dot-number) will be renamed to start with an `X`.

If the values are a simple table and are stored in a spreadsheet program such as Excel, and you're using R on Windows (as far as I can tell, integration of this feature on Mac and Linux isn't great), then there's a neat way to copy a table of data or a subset of one into R without saving the file into a new permanent file. (Don't worry if you can't find much information about this; it's seemingly undocumented.) The `file` argument to `read.table()` can, as you've seen, refer to a local file or a URL, but it can also be the keyword "`clipboard`", in which case any data you've most

recently “copied” from a spreadsheet or table-like text will be available, and an attempt will be made to read it in to memory. For instance, if you have some cells in a spreadsheet, such as those shown in figure 7.1, you can select and copy them to the system clipboard (click File > Copy or press Ctrl-C, depending on the system/program).

	A	B	C	D	E
1	height	weight	age	male	
2	151.765	47.82561	63	1	
3	139.7	36.48581	63	0	
4	136.525	31.86484	65	0	
5	156.845	53.04191	41	1	
6	145.415	41.27687	51	0	
7	163.83	62.99259	35	1	
8					
9					

**Figure 7.1** A selection of cells in a spreadsheet

Now you can use `read.table()` to read in this copied information:

```
## NOTE this may only work on Windows
read.table("clipboard", header = TRUE)
#>   height weight age male
#> 1 151.765 47.82561 63   1
#> 2 139.700 36.48581 63   0
#> 3 136.525 31.86484 65   0
#> 4 156.845 53.04191 41   1
#> 5 145.415 41.27687 51   0
#> 6 163.830 62.99259 35   1
```

**TIP** One package that automates this process more generally and also provides a way to paste in the code that creates an object is `datapasta`, available on CRAN. It does support most platforms, so it also works on Mac and Linux.

When you don’t want to save a different version of an Excel file, perhaps because it has several sheets that belong together, then there are options to connect directly to .xls and .xlsx files. Several packages have been written over the years to make this easier,

but the one I recommend most strongly is the `rio` package:

```
# install.packages("rio")
library(rio)
```

It can import more than 30 different file formats with a consistent syntax. `import()` is able to read in a wide variety of file formats:

```
excelContents <- import(file = "myExcelDocument.xls")
```

It can also optionally select a specific Excel sheet from which to import a table.

If data is in a less common format, then it's a good idea to research whether R is already able to read it, or otherwise search for packages that provide this support. There are very few file formats (beyond uncommon laboratory equipment that uses its own file format) that have zero support in R.

One of the largest annoyances people find with importing data into R is the assumptions made about what *type* the imported data is in. A CSV file doesn't store this information along with the data, so a row of data that looks like this

```
3, apple, 7.4, 2, banana
```

will be read in treating all of the numbers as type *numeric* (not *integer*). And because the default of `stringsAsFactors` is `TRUE`, the text values will be read in as type *factor*. Be careful about whether this is the result you want, and set `stringsAsFactors = FALSE` when importing data if it's not. The `import()` function from the `rio` package sets this automatically.

If data consists of lines to be read one at a time into values, then the `readLines()` function may be of use. This function, when the `con` (connection) argument is provided as a character string, opens the file of that name (if it can) and reads the requested number of lines (by default, all of them) in to the R session as a character *vector*. For example, if you have a file in the working directory named `myFile.txt` with the following contents

```
line one
line two
line three
```

then you can read this in to R with

```
readLines("myFile.txt")
#> [1] "line one"    "line two"    "line three"
```

**NOTE** `readLines()` will read the lines in as a character vector, regardless of the type of the data. If the data is not text, you will need to manually convert the result of `readLines()`.

There are many other ways to connect R to a data source, including connecting to databases (such as MySQL, SQLite, or Access), reading in a specific file type (myriad packages offer some sort of `read*( )` function), or connecting to a remote source. This last option warrants its own discussion because it can lead to great sources of data.

### 7.1.4 SCRAPING DATA

Sometimes you don't have your own source for data, but it is hosted somewhere else as a table on a web page, and you want to use the values for an analysis. You could copy out the table, format it, and save it to a file, but that involves a lot of manual processing that you would have to repeat if the data changed or you wanted to repeat the analysis.

**DEFINITION** *Web scraping* involves *scraping* (copying, collecting) content from the World Wide Web by using software to automatically extract the data from a web page's source file rather than just viewing it.

In that case, scraping the data may be a possibility.<sup>3</sup> If scraping is permitted, then we can tell R where to look on a web page to

read in some data. The `rvest` package provides useful functions to extract information out of the underlying HTML of a web page:

---

<sup>3</sup> Not all sites allow this, and it is highly advisable to check the usage policy of a site before attempting scraping.

---

```
# install.packages("rvest")
library(rvest)
# Loading required package: xml2
```

**DEFINITION** *Hypertext Markup Language (HTML)* is how web pages are constructed, using tags to change the appearance and layout of different elements. If you're not familiar at all with HTML but need to perform web scraping, it would be a good idea to do some reading up on HTML.

Walking through an example might make this clearer; let's scrape the table of milestones for R releases from Wikipedia, available at [https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language)). The table itself is rendered by a browser, as shown in figure 7.2.

[Milestones](#) [edit]

A list of changes in R releases is maintained in various "news" files at CRAN.<sup>[45]</sup> Some highlights are listed below for several major releases.

Release	Date	Description
0.16		This is the last <a href="#">alpha</a> version developed primarily by Ihaka and Gentleman. Much of the basic functionality from the "White Book" (see <a href="#">S history</a> ) was implemented. The mailing lists commenced on April 1, 1997.
0.49	1997-04-23	This is the oldest <a href="#">source</a> release which is currently available on CRAN. <sup>[46]</sup> CRAN is started on this date, with 3 mirrors that initially hosted 12 packages. <sup>[47]</sup> Alpha versions of R for Microsoft Windows and the <a href="#">classic Mac OS</a> are made available shortly after this version. <sup>[citation needed]</sup>
0.60	1997-12-05	R becomes an official part of the GNU Project. The code is hosted and maintained on <a href="#">CVS</a> .
0.65.1	1999-10-07	First versions of <code>update.packages</code> and <code>install.packages</code> functions for downloading and installing packages from CRAN. <sup>[48]</sup>
1.0	2000-02-29	Considered by its developers stable enough for production use. <sup>[49]</sup>
1.4	2001-12-19	S4 methods are introduced and the first version for <a href="#">Mac OS X</a> is made available soon after.
2.0	2004-10-04	Introduced <a href="#">lazy loading</a> , which enables fast loading of data with minimal expense of system memory.
2.1	2005-04-18	Support for <a href="#">UTF-8</a> encoding, and the beginnings of <a href="#">internationalization</a> and <a href="#">localization</a> for different languages.
2.11	2010-04-22	Support for Windows 64 bit systems.
2.13	2011-04-14	Adding a new compiler function that allows speeding up functions by converting them to byte-code.
2.14	2011-10-31	Added mandatory namespaces for packages. Added a new parallel package.
2.15	2012-03-30	New load balancing functions. Improved serialization speed for long vectors.
3.0	2013-04-03	Support for numeric index values $2^{31}$ and larger on 64 bit systems.
3.4	2017-04-21	Just-in-time compilation (JIT) of functions and loops to byte-code enabled by default.
3.5	2018-04-23	Packages byte-compiled on installation by default. Compact internal representation of integer sequences. Added a new serialization format to support compact internal representations.

**Figure 7.2 Table of milestones for R releases, as of September 2018**

You can save the URL for the Wikipedia page as a character string, to avoid typing it repeatedly:

```
r_lang_URL <-
"https://en.wikipedia.org/wiki/R_(programming_language)"
```

You can extract the HTML contents of the page into a special object with the `read_html()` function from `rvest`:

```
r_html <- read_html(r_lang_URL)
```

This opens a connection to the Wikipedia server and extracts the raw code the browser would otherwise interpret and present on a page. The content won't look like it makes much sense, but you'll process it a little and get out the bits you want. Don't worry if you get a warning along the lines of "Closing unused connection" at any point — it's just R cleaning up connections it no longer needs.

Let's extract all elements that have the HTML tag `table`:

```
html_nodes(r_html, "table")
#> [xml_nodeset (9)]
#> [1] <table class="infobox vevent"
```

```
>\n<caption class="s ...  
#> [2] <table  
class="wikitable">\n<tr>\n<th>Release</th>\n<th>Date</th>\n<t ...  
#> [3] <table class="plainlinks metadata ambox ambox-content ambox-Weasel" ...  
#> [4] <table class="plainlinks metadata ambox ambox-content ambox-peacock" ...  
#> [5] <table class="nowraplinks collapsible autocollapse navbox-inner" sty ...  
#> [6] <table class="nowraplinks collapsible autocollapse navbox-inner" sty ...  
#> [7] <table class="nowraplinks hlist collapsible collapsed navbox-inner" ...  
#> [8] <table class="nowraplinks navbox-subgroup"  
style="border-spacing:0"> ...  
#> [9] <table class="nowraplinks hlist navbox-inner"  
style="border-spacing: ...
```

The structure of these extractions is as follows, where . and # indicate a class and a named tag, respectively:

- x to select a tag element (<x>, for example <table>)
- .y to select a y class (<x class="y">, for example <div class="foo">)
- #z to select a named tag (<x id="z">, for example <p id="foo">)

Choosing View Source for the Wikipedia page in a browser and searching for *table* shows that the Milestones table is a table of class `wikitable`: <table class="wikitable">. You can select this part of the source with `html_node()` (since there is only one `wikitable` on the page):

```
release_html <- html_node(r_html, ".wikitable")
```

A period prepending a class is the selector you need here. The object returned from `html_node()` can be converted to a `data.frame`, at which point you have successfully read a table from a web page into a usable R structure:

```
release_table <- html_table(release_html)
```

Printing the first two columns (excluding the descriptions),

verifies that you have correctly extracted the table (comparing to the rendered result in figure 7.2):

```
release_table[, 1:2]
#>   Release      Date
#> 1    0.16
#> 2    0.49 1997-04-23
#> 3    0.60 1997-12-05
#> 4    0.65.1 1999-10-07
#> 5     1.0 2000-02-29
#> 6     1.4 2001-12-19
#> 7     2.0 2004-10-04
#> 8     2.1 2005-04-18
#> 9     2.11 2010-04-22
#> 10    2.13 2011-04-14
#> 11    2.14 2011-10-31
#> 12    2.15 2012-03-30
#> 13    3.0 2013-04-03
#> 14    3.4 2017-04-21
#> 15    3.5 2018-04-23
```

You could also extract the table of contents (TOC) using the `toc` tagged node (recalling that `#z` selects the tag named `z`):

```
tag_toc_html <- html_nodes(r_html, "#toc")
```

You can then extract the links of the unordered list (`<ul>`), each of which has a list item (`<li>`) tag and an anchor (`<a>`) tag:

```
toplevel_toc_links <- html_nodes(tag_toc_html, "ul li a")
```

Finally, you can extract the text from these links and view the result:

```
toplevel_toc_text <- html_text(toplevel_toc_links)
toplevel_toc_text
#> [1] "1 History"
#> [2] "2 Statistical features"
#> [3] "3 Programming features"
#> [4] "4 Packages"
#> [5] "5 Milestones"
#> [6] "6 Interfaces"
#> [7] "7 Implementations"
#> [8] "8 R Communities"
#> [9] "9 useR! conferences"
#> [10] "10 R Journal"
```

```
#> [11] "11 Comparison with SAS, SPSS, and Stata"
#> [12] "12 Commercial support for R"
#> [13] "13 Examples"
#> [14] "13.1 Basic syntax"
#> [15] "13.2 Structure of a function"
#> [16] "13.3 Mandelbrot set"
#> [17] "14 See also"
#> [18] "15 References"
#> [19] "16 External links"
```

The preceding walkthrough involved saving many intermediate variables that you likely won't need again. As you've seen, there is a better way to manage this pattern, and that's using chaining with `%>%` from the `magrittr` (or, if you're already using it, `dplyr`) package. The `rvest` package makes `%>%` available also:

```
r_lang_URL %>%      ①
  read_html() %>%    ②
    html_node(".wikitable") %>%    ③
      html_table() %>%    ④
        .[c(1, 2)]      ⑤
#>   Release      Date
#> 1 0.16
#> 2 0.49 1997-04-23
#> 3 0.60 1997-12-05
#> 4 0.65.1 1999-10-07
#> 5 1.0 2000-02-29
#> 6 1.4 2001-12-19
#> 7 2.0 2004-10-04
#> 8 2.1 2005-04-18
#> 9 2.11 2010-04-22
#> 10 2.13 2011-04-14
#> 11 2.14 2011-10-31
#> 12 2.15 2012-03-30
#> 13 3.0 2013-04-03
#> 14 3.4 2017-04-21
#> 15 3.5 2018-04-23
```

①

Takes the URL

②

Reads the HTML source

(3)

**Extracts the node with class wikitable**

(4)

**Converts the table to a data.frame**

(5)

**Selects the first two columns**

The TOC-scraping steps can be distilled down to the following:

```
r_lang_URL %>%
  read_html() %>%
  html_nodes("#toc") %>% ①
  html_nodes("ul li a") %>% ②
  html_text ③
#> [1] "1 History"
#> [2] "2 Statistical features"
#> [3] "3 Programming features"
#> [4] "4 Packages"
#> [5] "5 Milestones"
#> [6] "6 Interfaces"
#> [7] "7 Implementations"
#> [8] "8 R Communities"
#> [9] "9 useR! conferences"
#> [10] "10 R Journal"
#> [11] "11 Comparison with SAS, SPSS, and Stata"
#> [12] "12 Commercial support for R"
#> [13] "13 Examples"
#> [14] "13.1 Basic syntax"
#> [15] "13.2 Structure of a function"
#> [16] "13.3 Mandelbrot set"
#> [17] "14 See also"
#> [18] "15 References"
#> [19] "16 External links"
```

①

**Extracts the node with id toc**

②

**Extracts the ul, li, and a tags**

(3)

**Extracts the link text**

Clearly the pipe makes these steps much simpler and cleaner, so I strongly recommend building up scraping pipelines in this way. Add each step sequentially, checking that you are producing the structure you expect along the way.

### 7.1.5 INSPECTING DATA

You've already seen the `summary()` function, which gives an overview of an object when you have *continuous* data. These five-number summaries are useful for picturing how the data is distributed. When you have *discrete* data, however, things are a little different. If a column is character values, then you don't get a useful summary. When a column is a factor, `summary()` gives a count of the values in each level.

Another way to generate this that *does* work for character types is the `table()` function. When used on a factor type, it produces the same output as `summary()` (except it shows *all* levels, not just the first few):

```
table(iris$Species) ①
#>
#>     setosa  versicolor  virginica
#>     50        50        50
```

(1)

**The Species column of the iris dataset is of class factor.**

On a character type, it produces the same — a count of values in each unique level:

```
table(as.character(iris$Species))
#>
#>     setosa  versicolor  virginica
#>     50        50        50
```

Note that this will respect any differences such as upper- and lowercase, so `aBc` and `Abc` are counted as different:

```
table(c("aBc", "Abc", "aBc", "aBc", "Abc"))
#>
#> aBc  Abc
#> 3    2
```

When used on a numeric vector, the results become overwhelming as each different value is considered as its own level:

```
table(mtcars$mpg)
#>
#> 10.4 13.3 14.3 14.7   15 15.2 15.5 15.8 16.4 17.3 17.8
18.1 18.7 19.2 19.7
#> 2     1     1     1     1     2     1     1     1     1     1
1     1     2     1
#> 21   21.4 21.5 22.8 24.4   26 27.3 30.4 32.4 33.9
#> 2     2     1     2     1     1     1     2     1     1
```

When there are only a few unique values, this can still be very handy:

```
table(mtcars$carb)
#>
#> 1  2  3  4  6  8
#> 7 10  3 10  1  1
```

Sometimes it's just as useful to look at the top, bottom, or random values to see what's going on.

The `head()` function (which you've seen already, sprinkled throughout earlier chapters) prints the first (by default, six) rows of a `data.frame` (or similar, for example `matrix`, `tibble`, `table`) object, which is handy for checking the column names along with some of the data. It can be useful to check that there are actual values you expect to see in those rows (for example, not all `NA`, not all `0`, not all strings). For example:

```
head(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
Species
#> 1           5.1          3.5         1.4         0.2
```

```

setosa
#> 2      4.9      3.0      1.4      0.2
setosa
#> 3      4.7      3.2      1.3      0.2
setosa
#> 4      4.6      3.1      1.5      0.2
setosa
#> 5      5.0      3.6      1.4      0.2
setosa
#> 6      5.4      3.9      1.7      0.4
setosa

```

Similarly, the `tail()` function prints the lasts (by default, six) rows:

```

tail(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
Species
#> 145      6.7      3.3      5.7      2.5
virginica
#> 146      6.7      3.0      5.2      2.3
virginica
#> 147      6.3      2.5      5.0      1.9
virginica
#> 148      6.5      3.0      5.2      2.0
virginica
#> 149      6.2      3.4      5.4      2.3
virginica
#> 150      5.9      3.0      5.1      1.8
virginica

```

The number of rows to be printed can be controlled with the `n` argument, if you require more or fewer. For more of a peek throughout the data, randomly sample the rows and print those. There is a `dplyr` function that makes this easy:

```

set.seed(1)    ①
dplyr::sample_n(mtcars, 6)
#>   mpg cyl  disp  hp drat    wt  qsec vs
am gear carb
#> Merc 230      22.8    4 140.8  95 3.92 3.150 22.90  1
0     4    2
#> Merc 450SE    16.4    8 275.8 180 3.07 4.070 17.40  0
0     3    3
#> Fiat 128      32.4    4  78.7  66 4.08 2.200 19.47  1
1     4    1

```

```
#> Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0
1      5     2
#> Valiant           18.1    6 225.0 105 2.76 3.460 20.22  1
0      3     1
#> Pontiac Firebird 19.2    8 400.0 175 3.08 3.845 17.05  0
0      3     2
```

(1)

**Setting the seed value ensures that you will get this same random sample.**

To view an object in its entirety, the `view()` function (note the capital *V* in `view()`) is very handy. When passed a data argument, it opens a data viewer (in the Editor pane, so it has its own file-like tab) containing the data, converted to a `data.frame` (where possible). This works in both a terminal and RStudio, though the latter has better support for filtering, sorting, and displaying. Give it a try; `View(mtcars)` should show something like figure 7.3.

## 7.1.6 DEALING WITH ODD VALUES IN DATA (SENTINEL VALUES)

*Sentinel values*, also known as flag, trip, rogue, or signal values, are used in data entry to highlight certain records. Perhaps an impossibly high or low value will be pushed to 99 or -99, or a missing value will be encoded as 0 or -1.

	mpg	cyl	disp	hp	drat	wt	qsec
<b>Mazda RX4</b>	21.0	6	160.0	110	3.90	2.620	16.46
<b>Mazda RX4 Wag</b>	21.0	6	160.0	110	3.90	2.875	17.02
<b>Datsun 710</b>	22.8	4	108.0	93	3.85	2.320	18.61
<b>Hornet 4 Drive</b>	21.4	6	258.0	110	3.08	3.215	19.44
<b>Hornet Sportabout</b>	18.7	8	360.0	175	3.15	3.440	17.02
<b>Valiant</b>	18.1	6	225.0	105	2.76	3.460	20.22
<b>Duster 360</b>	14.3	8	360.0	245	3.21	3.570	15.84
<b>Merc 240D</b>	24.4	4	146.7	62	3.69	3.190	20.00

**Figure 7.3 Data viewer invoked with `view()`**

We've already covered having `NA` values in data, but this isn't the only value you need to be on the lookout for. Sentinel values typically arise as shortcuts in data entry. They attempt to bring attention to themselves by standing out as distinct from regular values. In records of local daily temperature, you may find some values of 999, which probably aren't an indication that the measuring station was on fire, but rather encode information that is hopefully recorded elsewhere, such as noting that the equipment was failing or the measurement was somehow invalid. This is an okay practice, as long as everyone in contact with the data is clear about what the values represent, and have the appropriate key to decipher this code.

Spotting these values is trivial if you know what they are ahead of time. Let's say you were working on quality control of the `mtcars` data and wanted to highlight that the horsepower (`hp`) of the Maserati Bora seemed excessively high, or wasn't measured correctly, or couldn't be trusted for whatever reason. It's not uncommon to have a placeholder value that sticks out. In this case, the processing might look like the following:

```
mtcars_qc <- mtcars      ①
mtcars_qc["Maserati Bora", "hp"]  ②
#> [1] 335

mtcars_qc["Maserati Bora", "hp"] <- 999  ③
tail(mtcars_qc)    ④
#>          mpg cyl disp hp drat    wt qsec vs am
gear carb
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1
5     2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1
5     2
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1
5     4
#> Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1
5     6
#> Maserati Bora   15.0   8 301.0 999 3.54 3.570 14.6  0  1
5     8
#> Volvo 142E      21.4   4 121.0 109 4.11 2.780 18.6  1  1
4     2
```

(1)

**Creates a copy of the mtcars data for modifying**

(2)

**Inspects the value of the hp column for the Maserati Bora (recall that row names can be used to subset rows)**

(3)

**Replaces the hp value for this row with a sentinel value**

(4)

**tail() prints the last six rows (remember, head() prints the first six).**

Given that you know how to subset data conditionally, you can look for known values specifically and select them. An example search might look like this:

```
mtcars_qc[mtcars_qc$hp == 999, ]      ①
#>          mpg cyl disp  hp drat    wt qsec vs am gear
carb
#> Maserati Bora 15     8   301 999 3.54 3.57 14.6  0  1     5
8
```

(1)

**Uses the base square-bracket method that preserves row names, whereas dplyr::filter() does not**

If you found some and wanted to remove those records, you could do so similarly:

```
tail(mtcars_qc[mtcars_qc$hp != 999, ])
#>          mpg cyl disp  hp drat    wt qsec vs am
gear carb
#> Fiat X1-9       27.3   4  79.0  66 4.08 1.935 18.9  1  1
4     1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1
5     2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1
5     2
```

```
#> Ford Pantera L 15.8    8 351.0 264 4.22 3.170 14.5  0  1
5     4
#> Ferrari Dino     19.7    6 145.0 175 3.62 2.770 15.5  0  1
5     6
#> Volvo 142E       21.4    4 121.0 109 4.11 2.780 18.6  1  1
4     2
```

You have more work to do when you aren't sure that any of these values exist in your data, but you can be diligent and check. A good sentinel value will be obvious in the context of the remaining data, so one way to check for these is to examine the distribution of values. You've seen how to do that already using the `summary()` function. From the summary, it can be easy to see that a `Min.` or `Max.` value stands out from the rest:

```
summary(mtcars_qc[, 2:5]) ①
#>      cyl          disp         hp
drat
#> Min.   :4.000   Min.   : 71.1   Min.   :52.0   Min.
:2.760
#> 1st Qu.:4.000   1st Qu.:120.8   1st Qu.:96.5   1st
Qu.:3.080
#> Median :6.000   Median :196.3   Median :123.0   Median
:3.695
#> Mean    :6.188   Mean    :230.7   Mean    :167.4   Mean
:3.597
#> 3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0   3rd
Qu.:3.920
#> Max.    :8.000   Max.    :472.0   Max.    :999.0   Max.
:4.930
```

①

**Just checking columns two through five.**

Note that the `Max.` value of `hp` is *much* larger than the `Mean` value.

`NA` itself is a sentinel value in that it encodes missingness. Other possible sentinel values include infinity (`Inf` or `-Inf`), which can arise when you divide a number by 0, or `NaN`, which tends to indicate  $0/0$ . Each of these can be captured by the `is.finite()` function, which returns `FALSE` for `NA`, `Inf`, `-Inf`, and `NaN`. If you combine this with the `any()` function (which returns `TRUE` if *any* of the values passed to it are `TRUE`), you can test for all

of these at once:

```
any(!is.finite(c(1, 2, 3, 4)))
#> [1] FALSE

any(!is.finite(c(1, 2, NA, 4)))
#> [1] TRUE

any(!is.finite(c(1, 2, NaN, 4)))
#> [1] TRUE

any(!is.finite(c(1, 2, Inf, 4)))
#> [1] TRUE
```

## 7.1.7 CONVERTING TO TIDY DATA

Once you've read data into R, it's time to do something with it. Often the data isn't in the structure that an analysis function requires. Most of the tidyverse expects the wide structure: one row per observation and one column per observable/measurable quantity. If data is in the long format and needs to be wide, you can easily do that with the `tidyverse` package:

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

You can use the `spread()` function to spread the data out into a wide format. Let's begin with the long-format data from the start of this chapter:

```
long_df
#> #>      id variable value
#> 1 person1   height 151.8
#> 2 person2   height 139.7
#> 3 person3   height 136.5
#> 4 person1   weight  47.8
#> 5 person2   weight  36.5
#> 6 person3   weight  31.9
#> 7 person1     age  63.0
#> 8 person2     age  63.0
#> 9 person3     age  65.0
#> 10 person1    male  1.0
#> 11 person2    male  0.0
#> 12 person3    male  0.0
```

You can then use `spread()` to create new columns using the variable column as the key to new column names, with values from the value column:

```
wide_df <- spread(data = long_df, key = variable, value = value)
wide_df
#>      id age height male weight
#> 1 person1 63 151.8    1   47.8
#> 2 person2 63 139.7    0   36.5
#> 3 person3 65 136.5    0   31.9
```

This is now in the wide format you require.

If you *do* need to go the other way, the `gather()` function from `tidyverse` gathers up a selection of columns and makes a long-format object:

```
long_df_again <- gather(
  data = wide_df,
  key = "variable", ①
  value = "value", ②
  -id ③
)
long_df_again
#>      id variable value
#> 1 person1     age 63.0
#> 2 person2     age 63.0
#> 3 person3     age 65.0
#> 4 person1   height 151.8
#> 5 person2   height 139.7
#> 6 person3   height 136.5
#> 7 person1     male  1.0
#> 8 person2     male  0.0
#> 9 person3     male  0.0
#> 10 person1   weight 47.8
#> 11 person2   weight 36.5
#> 12 person3   weight 31.9
```

①

The name to give to the column of variables

②

The name to give to the column of values

(3)

Selects which columns to gather, in this case all except the id column

## A base way to reshape data

There are good reasons to begin with this tidyverse way of chaining from long to wide data, and one of those reasons is that the base way of doing so is confusing and difficult.

The built-in stats package does have a `reshape()` function, but it's not nearly as elegant:

```
reshape(
  long_df,
  idvar = "id",
  timevar = "variable",
  direction = "wide"
)
#>      id value.height value.weight value.age
value.male
#> 1 person1      151.8      47.8      63
1
#> 2 person2      139.7      36.5      63
0
#> 3 person3      136.5      31.9      65
0
```

Notice that the column names have the prefix `value.`. The argument names are complex and difficult to recall.

The `reshape2` package makes this significantly easier. Here you load that package:

```
# install.packages("reshape2")
library(reshape2)
#
#> Attaching package: 'reshape2'
#> The following object is masked from
#> 'package:tidyverse':
#>
```

```
#>     smiths
```

The `tidyr` and `reshape2` packages both contain the `smiths` dataset, the former encoded as a `tibble`, and the latter as a `data.frame`. Loading and attaching the `reshape2` package *after* the `tidyr` package *masks* which version will become available if you enter `smiths` into the Console.

You can use the `dcast()` function, which performs the same task using a formula interface:

```
dcast(long_df, id ~ variable)
#>      id age height male weight
#> 1 person1  63  151.8    1   47.8
#> 2 person2  63  139.7    0   36.5
#> 3 person3  65  136.5    0   31.9
```

`tidyr` is built as a replacement to `reshape2`, so it's not surprising that it works as easily.

Other things to check include the following:

- Do you have/want characters or factors? Should you use `tibbles` or set the argument `stringsAsFactors`?
- What are the levels/unique values of your columns? Are there more or fewer than you expect? Do they correctly represent your data? Are there obvious typos?
- Do you have any missing values? Sentinel values? Do you expect any columns to have `NAs`? What about the range of present values?
- Do you have repeated rows/observations?

The last of these — repeated rows — occurs regularly. R will occasionally warn you when this is the case, but not always. To determine whether you have duplicate rows, use the `duplicated()` function from the `base` package. Suppose you have a `data.frame` with repeated measurements, say

```
rep_df <- data.frame(
```

```

name = c("x", "y", "y", "z", "x"),
val1 = c(3, 5, 5, 4, 8),
val2 = c(9, 2, 2, 1, 1),
val3 = c("q", "r", "r", "s", "t")
)
rep_df
#>   name val1 val2 val3
#> 1   x     3     9     q
#> 2   y     5     2     r
#> 3   y     5     2     r
#> 4   z     4     1     s
#> 5   x     8     1     t

```

You can ask, “Which rows are duplicates?” (appear more than once) with the following:

```
duplicated(rep_df)
#> [1] FALSE FALSE  TRUE FALSE FALSE
```

This returns a logical vector corresponding to which rows have already appeared, reading sequentially from row 1. Since row 2 has not appeared previously, it is not — at that point — a duplicate. Row 3, however, is the second appearance of these values, so it is a duplicate. Rows 1 and 5 both have x in the name column, but the other values make the row unique. If you reverse the search (from the last row) you see the same issue, in that there is only one duplicate:

```
duplicated(rep_df, fromLast = TRUE)
#> [1] FALSE  TRUE FALSE FALSE FALSE
```

You can therefore remove this duplicate row since you have a logical vector that you can negate to identify which rows to keep:

```
rep_df[ !duplicated(rep_df), ]
#>   name val1 val2 val3
#> 1   x     3     9     q
#> 2   y     5     2     r
#> 4   z     4     1     s
#> 5   x     8     1     t
```

Alternatively, there is a shortcut to this exact expression:

```
unique(rep_df)
#>   name val1 val2 val3
```

```
#> 1   x   3   9   q
#> 2   y   5   2   r
#> 4   z   4   1   s
#> 5   x   8   1   t
```

This function also works on vectors, so you can extract unique values from them:

```
unique(rep_df$val2)
#> [1] 9 2 1
```

## 7.2 Merging data

Having individual pieces of data stored in individual `data.frames` or `tibbles` makes each of them easy to access and work with, but there are times when you need to bring two of these objects together. This most often happens when there is overlap (say, a column of identifiers) between two that link each record. The separate objects may be consistent on their own and represent different things (this is a good way to store data), but to compare between the two properties, you need a way to join the objects.

Here is a `tibble` representing some countries and their capitals (which perhaps contain some data specific to your analysis):

```
# install.packages("tibble")
library(tibble)
countries_capitals <- tribble(    ①
  ~Country,      ~Capital,
  "Australia",   "Canberra",
  "France",      "Paris",
  "Malaysia",    "Kuala Lumpur",
  "India",        "New Delhi",
  "Nauru",        NA,      ②
  "Russia",       "Moscow",
  "Switzerland",  NA,      ②
  "Zimbabwe",    "Harare"
)
```

①

Creates a tibble row by row

(2)

### Nauru and Switzerland have no official capitals, only de facto capitals.

Here's another larger tibble representing the populations of capital cities around the world sourced from elsewhere (in this case, from <http://mng.bz/En5J> as of the end of 2017):

```
cities_populations <- tribble(
  ~City, ~Population,
  "New Delhi", 16787949L,
  "Moscow", 11541000L,
  "Paris", 2241346L,
  "Harare", 1487028L,
  "Kuala Lumpur", 1381830L,
  "Dublin", 1173179L,
  "Ottawa", 898150L,
  "Wellington", 405000L,
  "Canberra", 354644L
)
```

You can join these using one of `dplyr`'s join verbs. These borrow from the syntax of Structured Query Language (SQL), a language built for interacting with databases) and perform merging operations.

If you haven't already, install, load, and attach the `dplyr` library:

```
# install.packages("dplyr")
library(dplyr)
#
# The following objects are masked from 'package:stats':
#>
#>     filter, lag
# The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

The most straightforward option is to perform a *full join* of the two objects — take all the records from the first and match up with all the records from the second, joining on the column representing the capital city, ensuring that any missing records

from either are kept:

```
full_join(
  countries_capitals,      ①
  cities_populations,       ②
  by = c("Capital" = "City") ③
)
#> # A tibble: 11 x 3
#>   Country     Capital   Population
#>   <chr>       <chr>        <int>
#> 1 Australia  Canberra    354644
#> 2 France     Paris       2241346
#> 3 Malaysia   Kuala Lumpur 1381830
#> 4 India      New Delhi  16787949
#> 5 Nauru      <NA>         NA
#> 6 Russia     Moscow      11541000
#> 7 Switzerland <NA>         NA
#> 8 Zimbabwe   Harare     1487028
#> 9 <NA>        Dublin     1173179
#> 10 <NA>       Ottawa     898150
#> 11 <NA>       Wellington 405000
```

①

**The first object, containing the countries and their capitals**

②

**The second object, containing the cities (capitals) and their populations**

③

**Indicates the columns with common information are the Capital and City columns. You'll join the two objects by this column. If the column name(s) are common between the two objects, this doesn't need to be specified; otherwise the linkage is spelled out as shown here.**

Notice that every row from both the first and second objects is now present; where there was no information to merge in, the element is `NA`. Nauru and Switzerland have no official capitals, so their population columns remain as `NA`. Similarly, the `tibble` of countries and capitals didn't include Canada, Ireland, or New

Zealand, so the capitals of those (Ottawa, Dublin, and Wellington, respectively) have NA as their country.

It's important to note that when joining/merging two objects, only the information found in the row(s) containing the common value(s) between them will appear in the resulting object. The common value(s) must exactly (according to R) match for this to work. Let's try the preceding example with some not-quite-matching common values:

```
dataA <- tribble(
  ~name, ~value,
  "x", 1,
  "y", 2,
  "z", 3
)
dataB <- tribble(
  ~name, ~other_value,
  "x", 4,
  "Y", 5,    ①
  "z", 6
)
full_join(dataA, dataB)
# Joining, by = "name"
#> # A tibble: 4 x 3
#>   name   value other_value
#>   <chr> <dbl>     <dbl>
#> 1 x       1.00      4.00
#> 2 y       2.00      NA
#> 3 z       3.00      6.00
#> 4 Y       NA        5.00
```

①

Note that in this object, the second name is uppercase, whereas in the first object it's lowercase. It's not a match, and both y and Y will have rows in the object produced using a full\_join().

Joining two objects with a text string as the common value linking them is very common, but take care regarding the uniqueness of them. If a common value appears more than once, it will also be repeated in the resulting object:

```
dataA <- tribble(
```

```

~name, ~value,
"x", 1.1,
"x", 1.2,
"y", 2,
"z", 3
)
dataB <- tribble(
  ~name, ~other_value,
  "x", 4,
  "y", 5,
  "z", 6
)
full_join(dataA, dataB)
# Joining, by = "name"
#> # A tibble: 4 x 3
#>   name   value other_value
#>   <chr>  <dbl>     <dbl>
#> 1 x      1.10     4.00
#> 2 x      1.20     4.00
#> 3 y      2.00     5.00
#> 4 z      3.00     6.00

```

**CAUTION** Having non-unique common values becomes particularly important when you remember how R treats numbers — are `2L` and `2` the same in this context? As far as `dplyr` is concerned, they are, so you can in theory have numeric or integer values as the common link between two columns. Just be very cautious about how exact those values are, and check for uniqueness in the resulting object once you're done.

So far, you've seen the scenario where we want all the rows of the first object and all the rows of the second. What about when one object contains more information than you require? Returning to the population example, what if you *only* want the populations of capitals in the original object?

`dplyr` also has `left_join()` for situations like this; the resulting object will contain all the rows and columns of the first object (in a one-line call, the object on the left), and all the columns of the second object where there was a matching common value:

```

left_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 8 x 3
#>   Country     Capital     Population
#>   <chr>       <chr>        <int>
#> 1 Australia   Canberra      354644
#> 2 France      Paris         2241346
#> 3 Malaysia    Kuala Lumpur  1381830
#> 4 India       New Delhi    16787949
#> 5 Nauru        <NA>          NA
#> 6 Russia      Moscow        11541000
#> 7 Switzerland <NA>          NA
#> 8 Zimbabwe    Harare       1487028

```

Note here that this new object doesn't contain the rows that were only present in the `cities_populations` object, but all of those from `countries_capitals` (eight rows).

The complement to this is `right_join()` — the resulting object will contain all the rows and columns of the second object, and all the columns of the first object where there was a matching common value:

```

right_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 9 x 3
#>   Country     Capital     Population
#>   <chr>       <chr>        <int>
#> 1 India       New Delhi    16787949
#> 2 Russia      Moscow        11541000
#> 3 France      Paris         2241346
#> 4 Zimbabwe    Harare       1487028
#> 5 Malaysia    Kuala Lumpur  1381830
#> 6 <NA>         Dublin        1173179
#> 7 <NA>         Ottawa        898150
#> 8 <NA>         Wellington   405000
#> 9 Australia   Canberra      354644

```

This new object doesn't contain the rows that were only present in the `countries_capitals` object, but all of those from

`cities_populations` (nine rows).

An `inner_join()` produces an object that contains only the overlapping rows (it doesn't add rows of missing columns). This is very handy when you want to only work with the data you have values for:

```
inner_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 6 x 3
#>   Country   Capital     Population
#>   <chr>     <chr>        <int>
#> 1 Australia Canberra     354644
#> 2 France    Paris       2241346
#> 3 Malaysia  Kuala Lumpur 1381830
#> 4 India     New Delhi   16787949
#> 5 Russia    Moscow      11541000
#> 6 Zimbabwe Harare      1487028
```

Finally, there is a way to join using only rows that appear in one object but *not* in another: `anti_join()`. This is especially useful for determining which rows are missing from some object:

```
anti_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 2 x 2
#>   Country   Capital
#>   <chr>     <chr>
#> 1 Nauru     <NA>
#> 2 Switzerland <NA>
```

The preceding code produces an object with the rows from `countries_capitals` that don't appear in `cities_populations`; these countries have no `Capital` value, so they have no `city` value on which to match. If you want the complement to this, swap the first two arguments and rearrange the `by` argument:

```
anti_join(
  cities_populations,
```

```

countries_capitals,
by = c("City" = "Capital")
)
#> # A tibble: 3 × 2
#>   City      Population
#>   <chr>     <int>
#> 1 Dublin     1173179
#> 2 Ottawa     898150
#> 3 Wellington 405000

```

This produces an object with the rows appearing in `cities_populations` but missing from `countries_capitals`.

## A base way to merge data

base R has a way to merge `data.frames` too. The `merge()` function works similarly to `inner_join()` (with some slightly inconvenient quirks):

```

merge(
  countries_capitals,      ①
  cities_populations,
  by.x = "Capital",        ②
  by.y = "City"            ②
)
#>   Capital      Country Population
#> 1 Canberra    Australia  354644
#> 2 Harare      Zimbabwe 1487028
#> 3 Kuala Lumpur Malaysia 1381830
#> 4 Moscow       Russia   11541000
#> 5 New Delhi    India    16787949
#> 6 Paris        France   2241346

```

①

Note that since tibbles also have class `data.frame`, this works without any additional conversion.

②

The structure of the `by` argument is more specific in `merge()`.

To get something like `full_join()`, specify the `all`

argument:

```
merge(
  countries_capitals,
  cities_populations,
  by.x = "Capital",
  by.y = "City",
  all = TRUE
)
#>      Capital      Country   Population
#> 1    Canberra    Australia  354644
#> 2    Dublin      <NA>      1173179
#> 3    Harare      Zimbabwe 1487028
#> 4    Kuala Lumpur Malaysia 1381830
#> 5    Moscow      Russia   11541000
#> 6    New Delhi   India    16787949
#> 7    Ottawa      <NA>      898150
#> 8    Paris       France   2241346
#> 9    Wellington <NA>      405000
#> 10   <NA>        Nauru     NA
#> 11   <NA>        Switzerland NA
```

To get something like `left_join()`, specify just the `all.x` argument:

```
merge(
  countries_capitals,
  cities_populations,
  by.x = "Capital",
  by.y = "City",
  all.x = TRUE
)
#>      Capital      Country   Population
#> 1    Canberra    Australia  354644
#> 2    Harare      Zimbabwe 1487028
#> 3    Kuala Lumpur Malaysia 1381830
#> 4    Moscow      Russia   11541000
#> 5    New Delhi   India    16787949
#> 6    Paris       France   2241346
#> 7    <NA>        Nauru     NA
#> 8    <NA>        Switzerland NA
```

The biggest advantages of `dplyr`'s join functions over `base::merge()` are that they preserve the row order (`merge()` tends to move them around) and are substantially (potentially several times) faster because they're written in

C under the hood.

## 7.3 Writing data from R

After you've finished investigating your data (or even during that process), you'll eventually need to save the modified/cleaned/summarized data back to the disk. All the data you've been working with has so far only existed within the ephemeral memory of the R session.

There are several ways to save data, and just as with reading data *into* R, which one to use largely depends on your particular use case. The simplest way to save an R object to disk is with the `save()` function. This takes (via the `...` argument) some number of names of objects (as either bare names or character strings) to save (or a list of them via the `list` argument) and a filename (the `file` argument, optionally including the full path relative to the working directory, and typically ending with `.RData`). You can specify multiple objects in the `...` or `list` argument, and these will all be saved within the one `.RData` file:

```
z <- 3
save(mtcars, z, iris, file = "example_data_YYMMDD.RData")
```

The preceding code creates the file `example_data_YYMMDD.RData` in the working directory (or wherever you specify).

**NOTE** Remember to use sensible names for your files so that you will know what they represent later. For example, `mydata.RData` isn't particularly informative.

The major benefit of this method of saving data is that the data is saved preserving the R structure, not converted to text or flattened into a table. If you have a `tibble` with a list column and save it using `save()`, then reading it back in will restore that exact

object. Speaking of reading back in, an .RData file can be loaded back into a new session using the `load()` function, which takes a filename as the `file` argument:

```
load(file = "example_data_YYMMDD.RData")
```

If the preceding code is evaluated in a fresh, clean, empty R session, the Environment pane will now show the variables `mtcars`, `z`, and `iris`.

You can save *all* variables this way too. Listing each and every object you've created would be much too tedious, so thankfully there's a shortcut for this: `save.image()`. If that function sounds familiar, it may be because all the way back in chapter 1, I advised you to turn *off* the default option Save Your Workspace Image. This still stands; it's poor practice to automatically save your workspace, but there certainly may be times when you want to do it on purpose.

By default, `save.image()` saves to a file named `.RData` in the working directory (the filename beginning with a dot means it's treated as a hidden file in many operating systems), but you can name it whatever you like with the `file` argument. This image is created using `save()`, so it's loaded just the same as other `.RData` files, using `load()`.

If you only have a single object to save, an alternative is to use `saveRDS`, which behaves slightly differently. A single object can be specified using the `object` argument, and a filename (typically ending with `.rds` and optionally including the relative path) instructs where to save the file. In this case, I recommend using the object name in the name of the file, to assist with recalling what is saved, since the original object name won't be saved:

```
fruits <- c("strawberries", "apples", "watermelon")
saveRDS(fruits, file = "red_fruits.rds")
```

These files have a slight advantage over `.Rdata` files because they are compressed, so they take up less disk space.

Loading these in is slightly different. Use the `readRDS()`

function, which takes a `file` argument and instead of *loading* the object into the workspace, *returns* it:

```
readRDS(file = "red_fruits.rds")
#> [1] "strawberries" "apples"           "watermelon"
```

By itself, this just prints the object to the Console, but more usefully, it can be stored in a variable or used in further processing:

```
fruit_salad <- c("banana", "grapes", readRDS(file =
"red_fruits.rds"))
fruit_salad
#> [1] "banana"       "grapes"        "strawberries"
"apples"         "watermelon"
```

As you can see, the name of the new variable need not be the same as the one used when saving the object (or necessarily used at all), which can be both a blessing (when you need to save multiple copies of an object to files and then read them in) and a curse (when you fail to name the file correctly and can't recall what data it represents).

One last structure-preserving save mechanism: if you open a Stack Overflow question ([www.stackoverflow.com](http://www.stackoverflow.com)) because your code isn't working the way you expect it to, you'll often be (rightfully) asked to include some data that allows others to reproduce the issue you're facing. In this case, you want others to be able to take some of the data, apply your code, and dig into what's going wrong. If they don't have the right data, they can't be expected to encounter the same issue. In this case, sharing a saved file can be a bit difficult, so there's another way to output a structured object.

The `dput()` function writes out some R code that would generate the object passed to it. Here you have a `data.frame` with a few rows and columns of different types:

```
mixed_type_df <- data.frame(
  ints  = 1:4,      ①
  nums  = 2.5:5.5,  ②
  facts = factor(letters[23:26]), ③
```

```

  chars = state.abb[1:4],      ④
  stringsAsFactors = FALSE    ③   ④
)
mixed_type_df
#>   ints  nums  facts  chars
#> 1    1   2.5     w    AL
#> 2    2   3.5     x    AK
#> 3    3   4.5     y    AZ
#> 4    4   5.5     z    AR

```

①

**With this formatting, the sequence will be integers.**

②

**With this formatting, the sequence will be numeric.**

③

**To have some factor columns, despite stringsAsFactors=FALSE, specify them explicitly.**

④

**state.abb is a built-in dataset of the abbreviations of US state names. These will remain as characters due to stringsAsFactors=FALSE.**

You can see from the structure that the columns have different types, so you want to preserve that:

```

str(mixed_type_df)
#> 'data.frame':      4 obs. of  4 variables:
#> $ ints : int  1 2 3 4
#> $ nums : num  2.5 3.5 4.5 5.5
#> $ facts: Factor w/ 4 levels "w", "x", "y", "z": 1 2 3 4
#> $ chars: chr  "AL" "AK" "AZ" "AR"

```

For simple examples, sharing *that* code would be simplest, but this is only simple code because I've relied on sequences. The dput() function writes some R code to generate this structure:

```
dput(mixed_type_df)
```

```
#> structure(list(ints = 1:4, nums = c(2.5, 3.5, 4.5, 5.5),
  facts = structure(1:4, .Label = c("w",
#> "x", "y", "z"), class = "factor"), chars = c("AL", "AK",
  "AZ",
#> "AR")), .Names = c("ints", "nums", "facts", "chars"),
  row.names = c(NA,
#> -4L), class = "data.frame")
```

That looks like a mess, but it contains all R needs to know to rebuild `mixed_type_df`. If someone else wanted to reproduce that object, they could assign that code to `mixed_type_df` and know they're working with the same example data. Breaking up the lines into something readable and storing the result in a variable, it looks like the following:

```
mixed_type_df <- structure(
  list(ints = 1:4,
       nums = c(2.5, 3.5, 4.5, 5.5),
       facts = structure(
         1:4,
         .Label = c("w", "x", "y", "z"),
         class = "factor"
       ),
       chars = c("AL", "AK", "AZ", "AR")
     ),
     .Names = c("ints", "nums", "facts", "chars"),
     row.names = c(NA, -4L), class = "data.frame"
   )
mixed_type_df
#>   ints nums facts chars
#> 1    1   2.5     w    AL
#> 2    2   3.5     x    AK
#> 3    3   4.5     y    AZ
#> 4    4   5.5     z    AR
```

If the output from `dput()` seems too long, it's probably a sign that the example data is too large. Remember, if you're trying to highlight a particular issue you're facing, try paring it down to *just* the part of the data that seems to be involved in the issue. An extra 20 columns that are uninvolved in your analysis just get in the way of the question you're asking.

When you have tabular data and are likely to want/need to open it in a spreadsheet program (or other software that requires

such), it can be useful to write the data back to that format. You saw how to read in tabular data (such as .csv) files earlier, and sure enough there's a counterpart to that: `write.table()` and the convenience function `write.csv()`.

These take a single data object as an input and write a relevant file to a filename specified in the `file` argument (optionally, with a relative path). This is very handy for tabular structures such as `matrix` and `data.frame`. You could write the `mtcars` data set to a .csv file with the following:

```
write.csv(mtcars, file = "mtcars.csv")
```

By default, this writes the `row.names` to the file also. In the case of `mtcars`, this has benefit (since the names of the cars are stored there), but in general this will be a bit bothersome, so I usually turn it off with the argument `row.names = FALSE`.

There are other, more complex ways to save data; it is possible to write to Excel workbooks (.xls/.xlsx), although this often requires jumping through a few hoops. There are optimized methods for saving *large* data objects (see <https://blog.rstudio.org/2016/03/29/feather>). There are ways to communicate with databases — `dplyr` handles this quite well, abstracting away most of the database interaction. And there are of course many ways to write images to disk, but we'll talk about those when we discuss plotting.

## 7.4 Try it yourself

Try scraping the table of Australian City Populations found at <http://mng.bz/V7z5>. Hints: the table has the class `wikitable`, so it can be selected in the same way as the previous examples.

## Terminology

- *Working directory* — The directory in which R will look for files when reading/writing. Set with the `setwd()` function and queried with the `getwd()` function.

- *(Web) scraping* — Collecting data from an online resource using software to process the raw HTML source.
- *Sentinel value* — A blatantly incorrect value in a dataset, chosen to draw attention to some other meaning.
- *Long data* — When single observations are broken up between several rows involving the name of the observed measurement as an entry, we regard this as long data.
- *Wide data* — When a single row corresponds to a single observation, with values spread across columns denoting measured quantities, we regard this as wide data, and this conforms to the tidy principle.

## ***Summary***

- Long and wide data structures are different ways to store the same data.
- R has a working directory where it will start looking for (or try to store) files.
- Most plain-text stored data can be read in by R.
- R can read data from external sources, such as online.
- Web pages can be scraped directly into R objects using the `rvest` package.
- The `summary()` function can help reveal the presence of sentinel values.
- You can join two `data.frames` or `tibbles` with `dplyr`.
- Different join functions include different overlaps between tables.
- Multiple objects can be stored in an `.RData` file.
- Single objects can be efficiently stored in an `.rds` file.
- You can specify file locations relative to an RStudio Project with

the `here` package, or set the working directory yourself with `setwd()`.

- Path names in Windows use a backslash; make sure to escape this in strings referring to paths, or turn them around.
- Be aware of the separator used in your plain-text data files.
- Duplicated values in `duplicated()` only show up on their *second* occurrence.
- The base function `merge()` sorts rows alphabetically, whereas `dplyr`'s `*_join()` leaves them as they were.
- When writing out to a file, `row.names=FALSE` will prevent writing of `row.names` values.
- When saving objects, `save()` stores the name and contents of an object and is retrieved with `load()`. Using `saveRDS()` instead doesn't keep the name, so this needs to be assigned (if you want it) after retrieving with `readRDS()`.

# 8

## *Doing things conditionally: Control structures*

---

### **This chapter covers**

- Looping over sections of code
- Conditionally evaluating code
- Best practices to balance speed and readability

You saw in chapter 6 that you can conditionally select elements from data structures, but sometimes you only want to do (or not do) something based on some condition, or do something many times with slightly different conditions. There may be many steps you want to conditionally perform, so you need a way to only do things sometimes (or many times). You can use several approaches to do this in R, so let's look at the most common and the most useful, as well as some other ways these tasks can be accomplished.

### **8.1 Looping**

So far, you've evaluated code in a very linear manner, one line after another in a series of steps. This works fine as long as each step is a unique operation. Often, though, you want to repeat an operation several times (maybe a few times, maybe a large number of times). This comes up most frequently when you want

to perform some operation on each row of a `data.frame` one at a time. That usage flies in the face of the vectorized nature of R and should really be discouraged.

We'll still go through how to construct code that way, but you'll greatly benefit by thinking "vectorization" whenever you feel the urge to write code that performs a loop. To return that notion to the front of your mind, let's review what you learned in section 5.2.2.

### **8.1.1 VECTORIZATION**

You have already seen that, because R naturally uses vectors (operators know how to combine values from vectors and *recycle* values to a common length), you can use operators between two vectors (even if one has only a single element) and produce a vector output. Here's the example from chapter 5:

```
c(6, 5, 4, 5, 7, 4) > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

R recycles the smaller vector (5) to the same length as the longer vector (which it does whenever it can) and then performs the requested operation (`>` — the greater-than comparison) on each element-pair individually. Each comparison results in a `TRUE` or `FALSE` value, and together these are combined in a new vector.

Because most of these vectorized operations are performed at the C level rather than the R level (remember, many of the internal functions in R are compiled C or Fortran code), they are typically heavily optimized and complete very quickly.

If you had these values stored in a specific vector

```
test_nums <- c(6, 5, 4, 5, 7, 4)
```

you could (but shouldn't) write out each of the comparisons between each one-element subset and the value 5 and store the results in a vector:

```
is_test_nums_gt_5 <- c(
  test_nums[1] > 5,
```

```

test_nums[2] > 5,
test_nums[3] > 5,
test_nums[4] > 5,
test_nums[5] > 5,
test_nums[6] > 5
)
is_test_nums_gt_5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE

```

Whenever you see code that seems to be repeating itself over and over, it's usually a sign that you can perform some vectorization. Too often, though, it's taken as a sign that you need a loop over the repeated parts. You can compact the repetition by treating the part that changes as a sequence (and/or a vector), and perform all the comparisons using the following:

```

test_nums[1:6] > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE

```

To take full advantage of the vectorized nature of the operator (the function `>`), request that the entire vectors be compared this way:

```

test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE

```

There is a minor advantage to the explicit form in that you may want to alter the order in which these are compared.

Recall from the `areacircle()` example in section 4.1 that this pattern lends itself to *abstraction* — replacing values with some arbitrary placeholder and allowing it to change. This is the basis of *looping* over code in R, where the placeholder value is repeatedly changed for you.

There is a construct in R that does exactly this (“*for each of these placeholder values, do this...*”), but I strongly advise you to learn a more sophisticated way before you go down that path (which leads to poor practices and a nightmare for debugging). With that in mind, it's time to introduce the `purrr` package.

## **8.1.2 TIDY REPETITION: LOOPING WITH**

## PURRR

The `purrr` package (note the three r's) is another package I almost always load at the start of an analysis. Rather than just vectorizing a single operation, you can think of just about everything as a series of functions to apply repeatedly to elements.

**NOTE** Why three r's? Legend has it the package's author, Hadley Wickham, liked the “pure(r)” sound of this name and wanted the same number of characters as `dplyr` and other tidyverse-style packages (see <https://github.com/tidyverse/purrr/issues/35>). The more Rs the better, I say.

If you have installed the `purrr` package using `install.packages("purrr")` and loaded/attached it using

```
# install.packages("purrr")
library(purrr)
```

then what once might have been long and complicated repetition code is now easy to write.

The basic concept we'll focus on here is the `map()` behavior. In chapter 4, the function construct took in an argument and performed an operation on it. `map()` is a similar idea, but now you'll provide both the inputs and the function to which they will be passed, one at a time.

The object returned by `map()` will be a `list`, and it will *always* be a list.<sup>1</sup> A simple example of this is to use an existing function, say `sqrt()`, which takes the square root of its input:

---

<sup>1</sup> You'll see why this is an important point when we discuss the constructs that `map()` replaces.

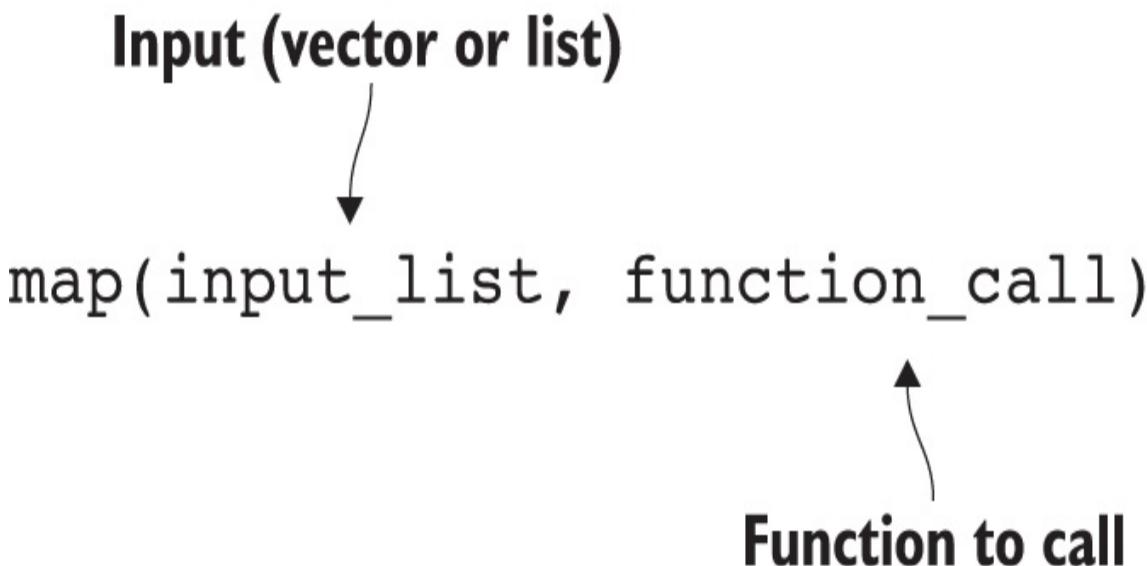
```
sqrt(9)
#> [1] 3
```

This is a trivial example, because as you've seen, `sqrt()` can take a vector as its argument and return a vector of square roots. This won't always be the case; the function you're working with can be arbitrarily complicated and won't necessarily work as intended with a vector argument.

To pass several values to this function one at a time (effectively *looping* over a vector of values), you can do so with `map()` by providing the "list" of values (an actual `list` or a vector) and the function to which they should be input:

```
map(c(4, 9, 16, 25), sqrt)
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] 3
#>
#> [[3]]
#> [1] 4
#>
#> [[4]]
#> [1] 5
```

The construction of calls to this function and the inputs are shown in figure 8.1.



**Figure 8.1** `purrr map()` construction

As with all the “tidy” functions, the data you’re using must be the first argument (formally named  $.x$  here).<sup>2</sup> The function argument (formally named  $.f$ ) is also provided, and you might note that it is not surrounded by quotes. This means you’re passing the function object itself as an argument (R is perfectly okay with this, as you saw in chapter 4). Additional arguments to the function can also be provided using the  $\dots$  mechanism (any arguments other than  $.x$  or  $.f$  will be passed along to the specified function when called).

---

<sup>2</sup> This means the pipe operator `%>%` will also work well with this function.

Now, a `list` as the return object might not be exactly what you want (though sometimes it might). As an added bonus, there are more specific `map_*` functions that have specific (*consistent*) return types. If instead of a `list` you want to return a numeric vector, you can do so with `map dbl()` (“map to double”), which is called the same way as before:

```
map_dbl(c(4, 9, 16, 25), sqrt)
#> [1] 2 3 4 5
```

If, instead, you want to return a character vector, it’s a matter of calling `map chr()` (“map to character”):

```
map_chr(c(4, 9, 16, 25), sqrt)
#> [1] "2.000000" "3.000000" "4.000000" "5.000000"
```

Note in this last example that the result is a character vector, created by first calculating the square roots of the values mapped over and then converting the (numeric) results to characters.

This becomes most powerful when the function applied to each list input returns a `data.frame` but you want the overall return type to be a single `data.frame`. Suppose you select two rows at random from the `mtcars` dataset using `dplyr`’s `sample_n()` function (with a specified *seed* value so you can get the exact same results on your machine):

```
set.seed(2)
dplyr::sample_n(tbl = mtcars, size = 2)
```

```
#>                               mpg cyl disp  hp drat    wt  qsec vs am
gear carb
#> Valiant          18.1   6 225 105 2.76 3.46 20.22  1  0
3   1
#> Dodge Challenger 15.5   8 318 150 2.76 3.52 16.87  0  0
3   2
```

To repeat this three times for three values of size, you can use the ordinary `map()`:

```
set.seed(2)
map(c(2, 3, 2), dplyr::sample_n, tbl = mtcars) ①
#> [[1]]
#>                               mpg cyl disp  hp drat    wt  qsec vs am
gear carb
#> Valiant          18.1   6 225 105 2.76 3.46 20.22  1  0
3   1
#> Dodge Challenger 15.5   8 318 150 2.76 3.52 16.87  0  0
3   2
#>
#> [[2]]
#>                               mpg cyl  disp  hp drat      wt  qsec vs am
gear carb
#> Honda Civic     30.4   4 75.7 52 4.93 1.615 18.52  1  1
4   2
#> Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0
3   1
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1
5   4
#>
#> [[3]]
#>                               mpg cyl disp  hp drat    wt  qsec vs
am gear carb
#> Maserati Bora    15.0   8 301 335 3.54 3.57 14.60  0
1   5   8
#> Hornet Sportabout 18.7   8 360 175 3.15 3.44 17.02  0
0   3   2
```

①

The `tbl` argument to `dplyr::sample_n()` is provided via the ... mechanism, whereas the `size` argument is looped over.

If you wanted just a `data.frame` as the result, you could use `map_df()` to join these outputs together, binding their rows on top of each other:

```
set.seed(2)
map_df(c(2, 3, 2), dplyr::sample_n, tbl = mtcars)
#> #>   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> 1 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> 2 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#> 3 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> 4 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> 5 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> 6 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
#> 7 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

**NOTE** The row names aren't output in the `map_df()` example. As a reminder, this is not uncommon in the tidyverse of packages, but it can lead to surprises if you expect it to work. The use of row names to store data is highly discouraged since it disconnects the metadata from the data. If these names are important, they should be in a column of their own. If you want to enforce that rule onto a `data.frame` with `row.names`, the `tibble` package has several functions for interacting with these, such as `rownames_to_column()`.

A slightly different version of this is the `walk()` function, which is useful when the function you're mapping over produces a *side effect* rather than returns a result. You may be producing a plot or printing something to the Console, and you don't want anything returned. For example, suppose you have a function that should only print a result:

```
print_value <- function(x) {
  print(paste("The value of x is ", x))
}
```

To loop over that, you *could* use `map()`:

```
ans_map <- map(1:3, print_value)
#> [1] "The value of x is  1"
#> [1] "The value of x is  2"
#> [1] "The value of x is  3"
```

```
ans_map
#> [[1]]
```

```
#> [1] "The value of x is 1"
#>
#> [[2]]
#> [1] "The value of x is 2"
#>
#> [[3]]
#> [1] "The value of x is 3"
```

But this returns the result into the variable. If you weren't assigning it, it too would `print()`. Instead, discard this returned data and use `walk()`:

```
walk(1:3, print_value)
#> [1] "The value of x is 1"
#> [1] "The value of x is 2"
#> [1] "The value of x is 3"
```

`walk()` returns the input in case you want to do more with it:

```
ans_walk <- walk(1:3, print_value)
#> [1] "The value of x is 1"
#> [1] "The value of x is 2"
#> [1] "The value of x is 3"

ans_walk
#> [1] 1 2 3
```

The function argument to `map()` (and related functions) is very flexible; instead of an existing function, you can also provide a *formula* that describes an unnamed (*anonymous*, sometimes called *lambda*) function.

**DEFINITION** *Anonymous function* means a function that's not assigned to a variable name. In some R functions, these can have a shorthand formula notation such as `~ .x + 1` to mean `function(.x) .x + 1`.

To just add 1 to the input list, you could do so without necessarily creating an `add_one()` function:

```
map_dbl(c(4, 5, 6), ~ .x + 1) ①
#> [1] 5 6 7
```

(1)

**The formula  $\sim .x + 1$  is equivalent to function(x)  $x + 1$ .**

You can write your own functions and refer to them by name. If you replace the “compare test\_nums to 5” operation with a function taking a general argument, you need a function that has a single argument (the index of the element to which the vector will be subset), so you could write the following:

```
compare_to_five <- function(value) {     ①     ②
  return(value > 5)     ③
}
```

(1)

**The name of the argument (value) refers to the value being compared to 5. This name is arbitrary.**

(2)

**The name of the function should reflect its action using a verb. In this case, the value being compared (5) is fixed, so it's included in the name.**

(3)

**An explicit call to return() isn't necessary, but it doesn't hurt to include it.**

This is a fairly trivial function, but it's of great benefit to think this way. Providing several inputs to this function (all the values in test\_nums) and returning a logical vector of the results is straightforward:

```
map_lgl(test_nums, compare_to_five)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Because the `>` operator already accepts vector input, so does the function (by extension):

```
compare_to_five(test_nums)
```

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

Again, with more complicated inputs and functions, this won't always be the case.

## The apply family

The family of `map` functions have older ancestors in base R, known as the `apply` family. These are `apply()`, `sapply()`, `lapply()`, and some others. They have the same goals as the `map` family but lack *type safety* in that the *type* of object returned is not guaranteed to be consistent with different inputs, which leads to some unexpected results.

Nonetheless, these show up often in published code so it's a good idea to recognize them.

For example, the `sapply()` function takes a vector and a function, and (most of the time) returns a vector of the results:

```
sapply(test_nums, compare_to_five)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Danger arises because `sapply()` can return several different types of object, depending on the input. base R does have a type-safe version, `vapply()`, but it requires you to specify the type each time you want to use it. You won't go wrong if you stick with the `map` family.

Although I strongly recommend you learn how to use these `purrr` functions, there *will* be times when you'll see more explicit (and discouraged) looping, so I also cover how that works next.

### 8.1.3 FOR LOOPS

In the previous example, you explicitly created a sequence of indices that identified which values to compare to the number 5. Let's replace the sequence 1, 2, 3, ... 6 with a placeholder, say, `i`

for index:

```
test_nums[i] > 5
```

You can perform this operation for each value  $i = 1, i = 2, \dots, i = 6$  using a *loop*. The simplest type of loop is called a `for` loop, which corresponds to the phrase “for the values of  $i$  in (some values), do (some expression).” Here’s the syntax of a loop:

```
for (i in values) {  
  expressions  
}
```

The layout of this is similar to when you wrote a function; you have the identifier `for` (similar to the `function` identifier), a statement in parentheses that instructs which values to use for  $i$  (similar to function arguments), and a set of expressions grouped together with curly braces (similar to the function body). When this is evaluated, R will assign the first value to  $i$  and evaluate expression, and then assign the second value to  $i$  and evaluate the expression again, and so on until it has assigned all of the values to  $i$ , at which point it will end evaluation.

## Loop scoping

Notice that you don’t have a return value when using a loop; the scoping rules that apply during loop operations are exactly the same as you had previously outside of functions. A loop instructs the program to repeat itself; it doesn’t create a new environment in which temporary values will be created, so you have no environment from which to return, and no values to return on completion.

Everything that happens during the loop evaluation happens in the environment in which it was called. This includes the assignment of values to the variable being looped (in the example,  $i$ ) — following the loop evaluation, that variable will retain its value:

```

for (i in 1:3) {
  ## do nothing
}
print(i) ## after the loop
#> [1] 3

```

For the earlier example comparing `test_nums`, you can *abstract* the index into a loop:

```

for (i in 1:6) {
  print(test_nums[i] > 5)
}
#> [1] TRUE
#> [1] FALSE
#> [1] FALSE
#> [1] FALSE
#> [1] TRUE
#> [1] FALSE

```

Note that this doesn't create a vector; it merely prints the values to the Console. This is an important distinction. To create a vector with these results instead, you can insert each result (since you're looping over the index), provided that the object exists first:

```

greater_than_5 <- c()      ①
for (i in 1:6) {
  greater_than_5[i] <- test_nums[i] > 5
}

```

①

**Creates an empty container in which the results will be stored**

There are several ways to create the initial empty object, including

```

c()
#> NULL

```

which creates an empty vector (containing no elements, so returns `NULL`), and

```
vector(length = 6)
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

which creates a (by default, logical) vector of the required length. You can also try this:

```
rep(NA, 6)      ①
#> [1] NA NA NA NA NA NA
```

①

**rep()** creates an object with its first argument repeated some number of times, specified by the second argument.

That last technique creates a vector of `NA` values of the required length. Each of these will work in the example as a vector into which values can be placed.

## On initial objects

If you decide you really need to do so, be very careful about how you create an initial empty vector into which to insert values. If for some reason a value can't be inserted, what will the default value be? Will `NA` or `FALSE` make sense for your object? Will type-coercion be an issue?

Another all-too-common pattern is to create a small initial object and grow it as needed. For example:

```
some_values <- c()
for (i in 1:100) {    some_values <- c(some_values,
i)  ①
}
```

①

**On each iteration, the `some_values` vector is copied into a vector one element larger.**

In this case, the vector `all_values` begins empty, and on each iteration it is copied into a slightly larger vector. This is

computationally expensive, and even this simple example takes a good few hundred thousands times longer than creating the vector once and *updating* the value at each index, like this

```
all_values <- rep(NA, 100)
for (i in 1:100) {
  all_values[i] <- i
}
```

despite producing the exact same result:

```
identical(all_values, some_values)
#> [1] TRUE
```

The looping variable can take any valid name, and it's still a good idea to name it carefully if it's meaningful. The values to be assigned to the variable you're looping over need not be such a simple sequence; any vector (or list) can follow the `in` identifier. You may want to loop over even values:

```
for (evens in c(2, 4, 6, 8)) {
  expression
}
```

Or you can loop over certain letters, this time using a list of values to iterate over:

```
for (vowel in list("a", "e", "i", "o", "u")) {
  expression
}
```

You can also generate the values as you go, using the result of another expression:

```
## 6 cylinder automatics in mtcars
for (auto_6cyl in row.names(mtcars[mtcars$cyl == 6 &
  mtcars$am == 1, ])) {
  print(auto_6cyl)
}
#> [1] "Mazda RX4"
```

```
#> [1] "Mazda RX4 Wag"
#> [1] "Ferrari Dino"
```

It should go without saying that this is a vastly inferior way (both in readability and evaluation speed) compared with the vectorized syntax. The few occasions on which this will be the structure you want to use include when one iteration depends on another, or when you really need to remain in the global scope.

## 8.2 Wider and narrower loop scope

You may wonder what the effect would be of changing the loop variable *inside* the loop. Let's try it and find out.

If you have a loop that prints out the value of the loop variable

```
for (i in 1:3) {
  message("starting i = ", i)
}

# starting i = 2
# starting i = 3
```

and you update *i* inside the loop, the value of *i* at the start of each iteration is still the loop value

```
for (i in 1:3) {
  message("starting i = ", i)
  i <- 7
}
# starting i = 1
# starting i = 2
# starting i = 3
```

but it takes the new value during the loop:

```
for (i in 1:3) {
  i <- 7
  message("ending i = ", i)
}
# ending i = 7
#> ending i = 7
#> ending i = 7
```

When you write loops using a function (as when using `purrr`'s

`map`), remember that any locally defined variables (within that function) won't persist outside of the looping.

They can, however, persist when you create a narrower scope within a loop, such as another loop (or a function, for that matter). If you (for whatever reason) want to loop over the rows and columns of a matrix, then in order to keep these iterators separate, they need different names. A logical choice is to name them based on that property. If you have a small matrix

```
m <- matrix(letters[1:6], nrow = 2, ncol = 3, byrow = TRUE)
m
#>      [,1] [,2] [,3]
#> [1,] "a"  "b"  "c"
#> [2,] "d"  "e"  "f"
```

then loop over both the rows and columns to print out some information, you can nest the loops knowing that the innermost calls will have access to the widest scope, so all of the iterators will be valid:

```
for (row in 1:2) {
  for (col in 1:3) {
    print(paste0("m[", row, ", ", col, "] = ", m[row,
col]))
  }
}
#> [1] "m[1, 1] = a"
#> [1] "m[1, 2] = b"
#> [1] "m[1, 3] = c"
#> [1] "m[2, 1] = d"
#> [1] "m[2, 2] = e"
#> [1] "m[2, 3] = f"
```

Keep an eye out for examples that match these patterns, and try to rewrite them in a vectorized format.

### 8.2.1 WHILE LOOPS

Another type of loop is the `while` loop. Rather than specify a list of values to loop over, a condition can be used; the loop will continue as long as the condition evaluates to `TRUE`. At the start of each loop iteration the condition is evaluated, and if it produces

TRUE, the loop body is executed. When the body has finished executing, the next iteration begins with another evaluation of the condition.

This requires that the condition be updated through the execution of the body — otherwise, the loop is infinite, and R will continue executing the loop until you explicitly cancel it (by using the stop button or pressing Esc):

1

**Always evaluates to TRUE**

If the condition *never* evaluates to TRUE, then the body is never executed:

```
while (1 == 2) {
    print("this line will not be printed")
}
```

The condition in the `while` statement needs to be valid, which means that any variables it depends on need to already exist. One way to deal with this is to be explicit about the first value by initializing it outside of the `while` statement. Within the body of the loop, the variable(s) can (and should) be updated. Here's a simple `while` loop that prints the current value of `i` on each iteration and then increases it by 1 until it is no longer less than 3:

```
i <- 0      ①
while (i < 3) {      ②
```

```

print(i)      ③
i <- i + 1  _____ ④
}
#> [1] 0
#> [1] 1
#> [1] 2

```

(1)

**Initializes i to 0**

(2)

**The condition of the loop is that i is less than 3, which it is to begin with.**

(3)

**On each iteration, the value of i is printed.**

(4)

**At the end of each iteration, i is increased by 1.**

## while loop scoping

As with the `for` loop construct, the operations are executed within the global scope, which means that any variables produced remain in the global scope once the loop is complete. In this case, the condition requires that the variables already exist because they need to be evaluated, but they remain as they were at the end of the loop execution:

```

i <- 1    ①
while (i < 4) {  ②
  i <- i + 1  _____ ③
}
print(i)    ④
#> [1] 4

```

(1)

Prior to the while loop, i has the value 1.

(2)

The condition here is that i is less than 4; it's 1, 2, or 3.

(3)

Within the loop body, i is increased by 1, even on the iteration where i starts with 3. Once this iteration is complete, the condition will be FALSE, and the loop will be exited.

(4)

After the loop, i has the value 4 because it was increased by 1 on the final evaluation iteration.

This construct can be very useful when you are repeating some operation on external varying data (scraping a website that changes, or monitoring an external condition). Alternatively, you may want to perform some operation(s) only when a specific condition is met.

## 8.3 Conditional evaluation

Rather than looping over values, a condition can be tested on its own and some code executed if it evaluates to TRUE. This is extremely useful but also requires great care. There are several ways to specify conditional evaluation depending on the result you are trying to achieve.

### 8.3.1 IF CONDITIONS

The most straightforward way to introduce a conditional evaluation into code is to use the `if` construct. Here you again test a condition, and if it evaluates to TRUE, a statement will be

executed:

```
x <- 5      ①
if (x > 3) { ②
  print("x is greater than 3") ③
}
#> [1] "x is greater than 3"
```

①

Sets the value of x to 5. This could be a value that isn't yet known.

②

Tests whether x is greater than 3; if so, evaluates the statement between the curly braces.

③

If the tested statement returns TRUE, this will be evaluated.

If the condition is FALSE, then the statement is ignored entirely, even if it contained invalid code (thanks to R's lazy evaluation mechanism):

```
x <- 5
if (x < 2) {
  print(object_that_doesnt_exist)
}
```

Be careful when testing missing logicals. The only values that the condition can validly evaluate to are TRUE and FALSE. If the condition evaluates to NA, an error is raised:

```
x <- NA
if (x > 3) {
  print("x is greater than 3")
}

#> Error: missing value where TRUE/FALSE needed
```

This gets messy when the condition depends on the result of further calculations that unexpectedly evaluate to NA, but those are scenarios you haven't planned for, and R doesn't know

whether you want to evaluate the statement or not.

The condition doesn't need to be an expression of its own; it can be a value that is TRUE or FALSE:

```
x <- TRUE
if (x) {
  print("do all the things")
}
#> [1] "do all the things"
```

An offshoot of this is when a value can be coerced to either TRUE or FALSE. Recall that this is true of many values, such as "TRUE", "T", and 1 evaluating to TRUE; and "FALSE", "F", and 0 evaluating to FALSE.

The braces are not *technically* required if the entire statement to be evaluated conditional on the test is a single statement:

```
x <- 5
if (x > 3)
  print("x is greater than 3")
#> [1] "x is greater than 3"
```

But that's highly dangerous because the restriction of “a single statement” can easily be forgotten and bugs introduced. In the following code, a second line was added to the `if()` construct during some changes:

```
x <- 5
if (x > 3) ①
  print("x is greater than 3")
  print("this should only appear when x > 3") ②

x <- 2
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3") ③
```

①

Neglecting to use the braces is valid if you have only a single statement following the `if()`, but it's dangerous.

(2)

An earlier draft of this `if()` code had the single condition, but later a second one was introduced. It appears to still work fine when the condition is met.

(3)

When the condition isn't met, though, this line should not be evaluated — but it is, because it's not actually part of the `if()` construct.

Evaluating the first part doesn't highlight any issues; the condition is met, so both statements are printed:

```
x <- 5
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3")
#> [1] "x is greater than 3"
#> [1] "this should only appear when x > 3"
```

With the second construction, though,

```
x <- 2
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3")
#> [1] "this should only appear when x > 3"
```

the second message is printed (since it's not part of the `if()` construct at all), which was not the intention.

Sometimes you want to perform some other calculation when a condition returns FALSE. Rather than having to perform the test twice (for both `if (TRUE)` and `if (FALSE)`), an `else` statement can be added to this construct:

```
x <- 5
if (x > 3) {
  print("x is greater than 3")
} else {
  print("x is less than 3")
}
```

```
#> [1] "x is greater than 3"
```

The `else` needs to immediately follow the closing curly brace of the `if` body for the construct to be valid. The curly braces themselves are still not required if the body in both cases consists of a single line of commands or statement. The entire construct can be written on a single line if you want, though this quickly becomes unreadable if the code body is anything but simple:

```
x <- 5
if (x > 3) print("x is greater than 3") else print("x is
less than 3")
#> [1] "x is greater than 3"
```

You can even have more than one level of testing by adding *another if* statement to the `else`:

```
x <- 5
if (x > 6) {
  print("x is greater than 6")
} else if (x > 4) {
  print("x is greater than 4")
} else {
  print("x is less than 5")
}
#> [1] "x is greater than 4"
```

Note that only the "x is greater than 4" result was printed — once a condition is tested and found to be `TRUE`, evaluation exits the `if()` construct entirely, and the other conditions are not evaluated. This means they don't even need to be valid, because they will be lazily evaluated:

```
x <- 5
if (x > 3) { ①
  print("x is greater than 3")
} else if (nonsense) { ②
  print("how did I get here?")
}
#> [1] "x is greater than 3"
```

①

This condition evaluates to TRUE, so its statement will be evaluated.

(2)

nonsense isn't a defined variable, but it's not evaluated in this case. See what happens when the statement is evaluated by changing the value of x and evaluating the block again.

These conditions are tested in the order they're written, and once one evaluates to TRUE, that condition's corresponding code will be executed and the testing concluded.

## On overlapping conditions

Even if some of the conditions have overlap, the testing will be completed in order and either zero or one of the code blocks will be executed (not multiple blocks). For example, if you test whether  $x = 3$  is greater than 1 and greater than 2 (both being TRUE), then the order in which the tests are performed will still determine which code block is executed:

```
x <- 3
if (x > 2) {
  print("x is greater than 2")
} else if (x > 1) {
  print("x is greater than 1")
} else {
  print("x is smaller than 1 and 2")
}
#> [1] "x is greater than 2"
```

There is no limit to how many else statements are chained, but only zero or one of them will ever result in the corresponding code being executed.

The code block that is potentially executed following an if condition can contain any code, be that assignment (to the global environment), function calls, or a long series of calculations. An

even more useful feature of the `if` construct is that it can itself be used to conditionally assign something to a variable:

```
x <- 5
y <- if (x < 3) "small" else "big"
y
#> [1] "big"
```

That's equivalent to moving the assignment into the body of the statement, but it requires two separate assignments:

```
x <- 5
if (x < 3) {
  y <- "small"
} else {
  y <- "big"
}
y
#> [1] "big"
```

The `if` construct is designed to perform a single test; that is, the condition should return a single value of either `TRUE` or `FALSE`. You can construct conditions that return multiple logical values, but these shouldn't be used in the `if` construct condition. Doing so produces a warning:

```
if (c(1, 2, 3) > 2) { print("TRUE") }
# Warning message:
# In if (c(1, 2, 3) > 2) {: 
#> the condition has length > 1 and only the
#> first element will be used
```

The warning comes even though the first element is still accepted and used as the result of the test. In future versions of R, this will be an error because multiple values have no use in this construct.<sup>3</sup>

---

<sup>3</sup> You can enforce the error in recent versions of R by setting the environment variable `R_CHECK_LENGTH_1_CONDITION` to `true`.

To perform multiple conditional comparisons, a more suitable construct is available.

### 8.3.2 IFELSE CONDITIONS

An important feature of the `if` construct is that the condition *should* be of length 1, but what if you want to do something to each *element* of a vector depending on how it satisfies a condition? The `ifelse()` function is made for exactly this situation.

The arguments are structured similarly to the `if` and `else` identifiers. Specify a condition to test (which can return multiple values) and replacements for when the result is `TRUE` or `FALSE`, as in the following:

```
ifelse(test, yes, no)
```

`test` is either a logical vector or a computation that produces a logical vector (or can be coerced to one); and `yes` (and `no`) are vectors of values to be used when `test` returns `TRUE` (or `FALSE`). These two outcomes need not be the same type or even length, but exactly what will be returned is highly dependent on the value of `test`. The value produced by `ifelse()` is just another value, and can be assigned to a variable.

**NOTE** Great care needs to be taken with this function. The type of value returned depends on whether the test results in `yes` or `no` being evaluated.

You can call this function with two very different return options:

```
ifelse(TRUE, "a string", 7L)
#> [1] "a string"
```

```
ifelse(FALSE, "a string", 7L)
#> [1] 7
```

There is a third option; the test can result in `NA`, in which case the result will also be `NA`:

```
ifelse(NA, "test was TRUE", "test was FALSE")
```

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

That can lead to issues if the value of test changes and thus alters the type of the return value.

Due to lazy evaluation, if the test never results in one or both of yes or no, then R will never attempt to evaluate those arguments. That means you don't have to worry about undefined values if they're not going to be needed:

```
ifelse(FALSE, undefined_variable, 8L)
#> [1] 8
```

The test argument of `ifelse()` can be a vector of length greater than 1 (compare this to the test in an `if()` construct that *should* be of length 1 and produces a warning otherwise), and a consequence of this is that the result will be of that length. This makes sense; each element is tested, and either yes or no arguments or NA will be evaluated for each result. The arguments yes and no, though, can be of any length (either shorter or longer than test), but only the corresponding values will be used (or the result will be *recycled*):

```
ifelse(TRUE, c(2, 3), "z") ①
#> [1] 2
```

```
ifelse(c(TRUE, TRUE), 3, "z") ②
#> [1] 3 3
```

①

**Here the yes argument is longer than test, so only the first value is used.**

②

**Here the test argument is longer than yes, so yes is recycled.**

**CAUTION** If any element of either yes or no *will* be needed, then the entire vector must be valid (contain existing variables). When test is a vector, the corresponding elements from yes and no

will be extracted, but those vectors will need to be evaluated before that happens.

For example, here you'll extract the first value of yes and the second of no (both 3 in this example):

```
ifelse(c(TRUE, FALSE), c(3, 9), c(9, 3)) ①
#> [1] 3 3
```

①

**The test is TRUE for the first element, so it returns the first element of yes. It is FALSE for the second element, so it returns the second element of no.**

When the yes vector can't be constructed, though (because a variable is undefined), it fails:

```
ifelse(c(TRUE, FALSE), c(3, not_a_var), c(9, 3))
#> Error: object 'not_a_var' not found
```

This becomes particularly important when you want to perform an operation on a vector conditionally based on the specific values. If you have a vector that has both positive and negative values

```
x <- -3:3
x
#> [1] -3 -2 -1  0  1  2  3
```

you can try to take the square root of this:

```
sqrt(x)
# Warning message:
# In sqrt(x): NaNs produced
#> [1]      NaN      NaN      NaN 0.000000 1.000000
1.414214 1.732051
```

The warning is because you can't take the square root of a negative number, so the result contains `NaN` (not a number). This seems like a good candidate for `ifelse()`. You only want to take

the square root of  $x$  when the value is non-negative. One option would be to use  $x \geq 0$  ( $x$  greater than or equal to 0) as the test and `sqrt(x)` as the yes argument, since that's the result you want when the test is `TRUE`. You could use `NA` as the no argument so you don't get `NAN` there. When you try that, you get the result as expected (`sqrt(x)` for non-zero  $x$  and `NA` otherwise) but still generate the warning about `NaN`:

```
ifelse(x >= 0, sqrt(x), NA)
# Warning message:
# In sqrt(x): NaNs produced
#> [1]      NA      NA      NA 0.000000 1.000000
1.414214 1.732051
```

The reason is that `sqrt(x)` is still evaluated for all elements; you're just keeping the ones that make sense in the result. A better way to achieve this is to *only* pass sensible values to `sqrt()` by performing the `ifelse()` step first:

```
sqrt(ifelse(x >= 0, x, NA))
#> [1]      NA      NA      NA 0.000000 1.000000
1.414214 1.732051
```

That's because `sqrt(NA)` doesn't produce a warning (it just returns `NA`).

You can combine this `ifelse()` construct with `dplyr` operations because it's a vectorized operation. To create a new column in a `data.frame` where the value depends conditionally on some other column, use `mutate()` and `ifelse()`:

```
head(①
  dplyr::mutate(
    mtcars, ②
    hp_per_cc_auto = ifelse(am, hp / disp, NA) ③
  ) ④
)

#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb
hp_per_cc_auto
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4     4
0.6875000
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4     4
0.6875000
```

```
#> 3 22.8   4   108   93 3.85 2.320 18.61   1   1   4   1
0.8611111
#> 4 21.4   6   258 110 3.08 3.215 19.44   1   0   3   1
NA
#> 5 18.7   8   360 175 3.15 3.440 17.02   0   0   3   2
NA
#> 6 18.1   6   225 105 2.76 3.460 20.22   1   0   3   1
NA
```

(1)

**Only print() the first six rows.**

(2)

**Modifies the mtcars data.frame using dplyr**

(3)

**Creates a new column hp\_per\_cc\_auto, referencing columns from mtcars using nonstandard evaluation**

(4)

**Uses am (does the vehicle have an automatic transmission?) as the test; 1 indicates TRUE and is coercible. Where the test value is TRUE, divides hp by disp; otherwise returns NA.**

Passing other objects (for example, a list or matrix) is possible, but take note that the condition that the result be the same size and shape as test will remain.

There is a handy `dplyr` way to achieve this task: the `case_when()` function lets you use multiple conditions to produce different results and works inside a `mutate()` call. As shown next, when using `case_when()` inside `mutate()`, you can use column names without prefixing them with the name of their `data.frame`:

```
head(
  dplyr::mutate(
    mtcars,
    hp_per_cc_auto = dplyr::case_when(am == 1 ~ hp /
  disp, _____②_____
```

```

am == 0 ~
NA_real_) ③ ④
)
)
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb
hp_per_cc_auto
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
0.6875000
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
0.6875000
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
0.8611111
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
NA
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0    3    2
NA
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0    3    1
NA

```

②

The syntax for conditions is a comma-separated set of tests and results written as a formula: <test> ~ <result>.

③

All results must explicitly be the same type, so rather than use NA, you need the explicit NA\_real\_.

④

As a catch-all, the last condition can be TRUE ~<result>.

## 8.4 Try it yourself

Let's work through an example together. Say you have the unique values of cyl from the mtcars object:

```

## find the unique values of cyl in mtcars and sort them
numerically
## (by default they are in the order in which they first
appear)
car_cyls <- sort(unique(mtcars$cyl))

```

You want to produce some summary output of the most efficient (in terms of miles per gallon) vehicle in each cylinder class. You could produce an object of these (using `dplyr`), but let's make a function that writes a summary to the screen. Read through the following function to make sure you understand what it's doing:

```
## takes input num_cyls and prints a statement
## about mtcars rows with that number of cylinders
showcase <- function(num_cyls) {

  ## subset mtcars to the rows with num_cyls cylinders
  ## this uses [ instead of dplyr::filter since the latter
  ## drops row names
  cars_with_cyls <- mtcars[mtcars$cyl == num_cyls, ]

  ## identify the row with the highest mpg
  ## uses [ to preserve rownames
  ## which.max finds the index of the highest value
  most_efficient <-
  cars_with_cyls[which.max(cars_with_cyls$mpg), ]

  ## print a nice message to the console explaining the
  ## results
  cat(paste("There are", nrow(cars_with_cyls), "cars with",
  num_cyls,
            "cylinders.\nThe most fuel efficient of these
  is the",
            rownames(most_efficient), "\n\n"))

}

}
```

You can test this function with an example:

```
showcase(4)
#> There are 11 cars with 4 cylinders.
#> The most fuel efficient of these is the Toyota Corolla
```

Now, to produce this output for each of the classes in `car_cyls`, you might think to loop over those values. Instead, you can have `purrr` iterate over them:

```
# install.packages("purrr")
library(purrr)
## walk is equivalent to map but doesn't return anything
## explicitly
##   (actually returns the input list, invisibly)
```

```

walk(car_cyls, showcase)
#> There are 11 cars with 4 cylinders.
#> The most fuel efficient of these is the Toyota Corolla
#>
#> There are 7 cars with 6 cylinders.
#> The most fuel efficient of these is the Hornet 4 Drive
#>
#> There are 14 cars with 8 cylinders.
#> The most fuel efficient of these is the Pontiac Firebird

```

## ***Terminology***

- *Loop* — A repetition of some code, often replacing a particular value each time around.
- *Abstraction* — Replacing a specific value with a more general placeholder.
- *Side effect* — Something that happens that isn't directly related to the data involved, such as producing a plot, interacting with an external system, or printing something to the Console.
- *Type safety* — The notion that the return value(s) of a function will always be of an expected type. The lack of type safety of some functions makes working programmatically with data complex.

## ***Summary***

- Vectorization is a superior way to loop through elements of a structure.
- The `purrr` package's `map()` function (and associates) makes repetition clean, safe, and fast.
- Anonymous functions can save having to explicitly create a function.
- Variables created during explicit loops remain in the current scope.
- Conditional evaluation can be achieved using `if()` and `ifelse()`.

- Loops can often be replaced with faster and simpler vectorization.
- Temporary (anonymous) functions can be created inside `map` using a formula: `~ .x + 1` is shorthand for `function(x) x + 1`.
- Variables created within loops are still in the global namespace after evaluation of the loop is complete.
- Conditions used in calls to `if()` should only be a single element (length 1).
- The result of a call to `ifelse()` will be the same size and shape as the condition tested, with values taken from the yes or no arguments.

# 9

# *Visualizing data: Plotting*

---

## This chapter covers

- Preparing data for visualizations
- Plotting data the tidy way: `ggplot2`
- Saving graphics to files

Visualizations of data help both you and people you share your results with gain insight into relationships inherent in the data. When you have many data points (even just a handful), a *plot* (a graph, a graphic, or some other sort of image) can convey a great deal of information that a table can't. That's not to say plots are *always* the best way to convey information, but when they are, you need tools to generate them easily and effectively with R.

### **9.1 Data preparation**

By now you've taken raw data, performed some operations on it (cleaned, sliced, and diced), and are ready to plot it to see what secrets it might be able to tell. The optimum arrangement of the data in terms of joining together pieces of information may not, however, be the optimum arrangement in terms of thinking about reslicing it for visualization.

It's not uncommon to need to generate another version of a dataset specifically for visualization, but that should be

reproducibly derivable from both the original raw data and any subsequent transformations performed. When you have various groups in data, one useful way to include this information in a visualization is to visually separate the groups, by color, shape, linetype, or physical distance. This means the data needs to be in a format that is amenable to such splitting.

### **9.1.1 TIDY DATA, REVISITED**

We've discussed *tidy data* a few times now, and visualizations will lead to yet another justification for maintaining these principles in your data. The idea of wide data with one row per subject and one column per observation means that grouping together observations with something in common (for example, all the automatic transmission vehicles) becomes as simple as filtering values of one column.

If data is in the long data format (few columns, and a variable describing which observation a measurement refers to), then this filtering becomes much more complex. If you want to group observations, you need to filter both the column that describes variables and the column of measurements. This quickly becomes unwieldy. Tidy data is therefore a much more suitable format for visualization.

### **9.1.2 IMPORTANCE OF DATA TYPES**

We covered the different data types early on, and perhaps now you've seen examples where the specific type in which a column of data is stored alters the analysis performed on it. This becomes particularly important when filtering values, which is an integral part of subsetting and grouping.

Many of the visualization tools you're about to see are as easy to use as they are because they automate a lot of common tasks. When a data type is *continuous* (can take any value, not just some discrete categories) the plots will reflect this quality. When you do need categories, however, the plots produced will be constructed differently, in a way that reflects this quality instead.

You won't (necessarily) need to tell R to use a different tool for these different types; the tools have been *abstracted* to allow either, and do something different depending on which data type is present. This behavior can be overridden, which you'll also see.

You should then place high importance on ensuring that data is in the correct format before you attempt to visualize it. Does the quantity you want to use as the *grouping variable* — the quantity that defines which group a measurement belongs to, in order to somehow distinguish these in a plot — relate to a continuous value (say, numeric) or a discrete value (a factor)? These two options produce visually different plots.

## 9.2 *ggplot2*

Many different visualization tools are available for R, but one of the most popular and sophisticated is `ggplot2`. This is the second incarnation of the “grammar of graphics” plotting system developed by Hadley Wickham as an implementation of Leland Wilkinson's *Grammar of Graphics* (Springer, 2005).

The general concept that this package encapsulates is that of a *grammar*, an explicit language that connects *data* (the first and foremost feature in a visualization) to what you see on the screen as *aesthetic mappings*. In a spreadsheet, you would perhaps select the values to have on the x-axis and y-axis, produce a plot, and then adjust colors and settings to achieve the plot you want. In `ggplot2`, these settings are directly connected to the data through the *mappings* — for example, when plotting columns from `mtcars`, determining the color of points by whether or not the vehicle had an automatic transmission, as defined by corresponding values in the column `am`. The other primary features of `ggplot2` are the types of features you want to plot (*geometries*) and the relationships between values within a group or aspect (*scales*).

These explicit mappings mean you have more control over how plots are generated, but with such sophistication you have a syntax to become accustomed to.

## 9.2.1 GENERAL CONSTRUCTION

The four features I mentioned — data, geometries, aesthetic mappings, and scales — need to be presented to the `ggplot2` code in a particular way. For better or worse, `ggplot2` takes full advantage of the non-standard evaluation framework you saw in `dplyr`, which means you can refer to columns of a `data.frame` or `tibble` in isolation within a `ggplot2` call.

Begin by installing, loading, and attaching the `ggplot2` package:

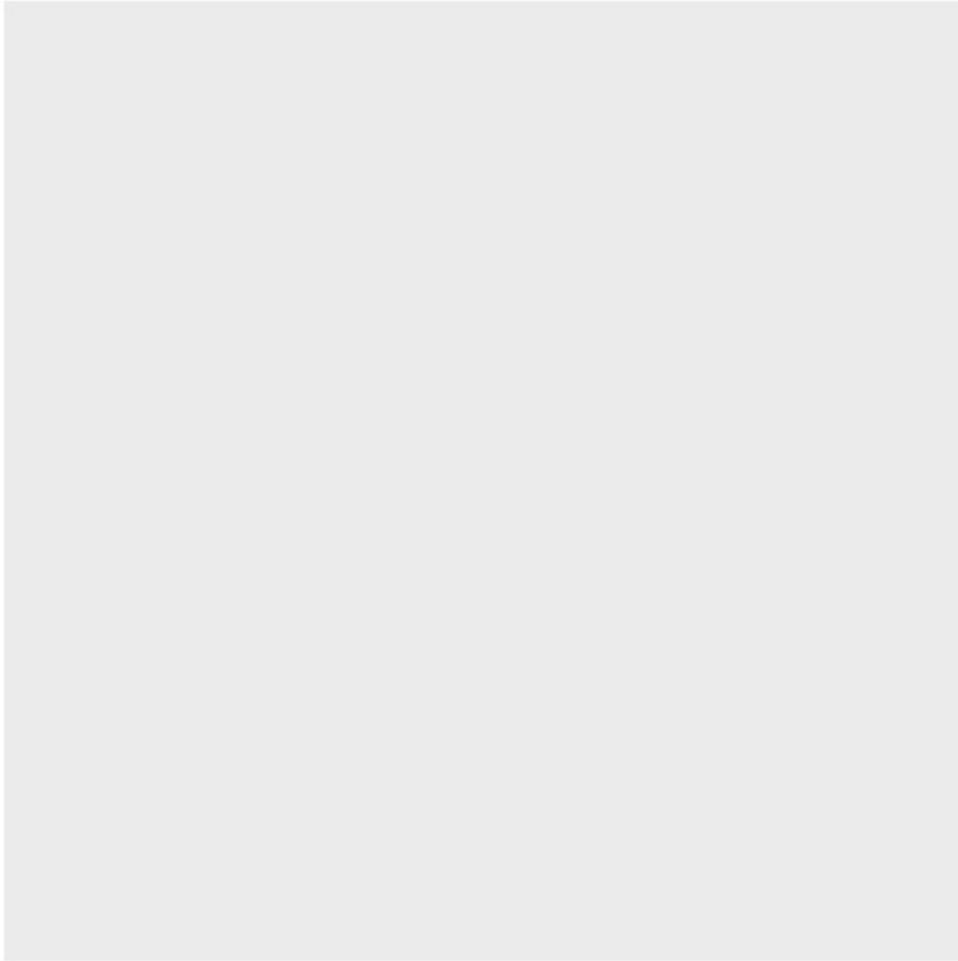
```
# install.packages("ggplot2")
library(ggplot2)
```

The first ingredient to constructing a `ggplot2` visualization is to request one with the `ggplot()` function:

```
ggplot()
```

Doing so with no further information produces an empty plot in the Plot tab, as shown in figure 9.1.

Note, though, that the plot area isn't just empty; a grey background is present. Depending on how the panes are set up in RStudio, the width and height of this may be different than what is shown here.



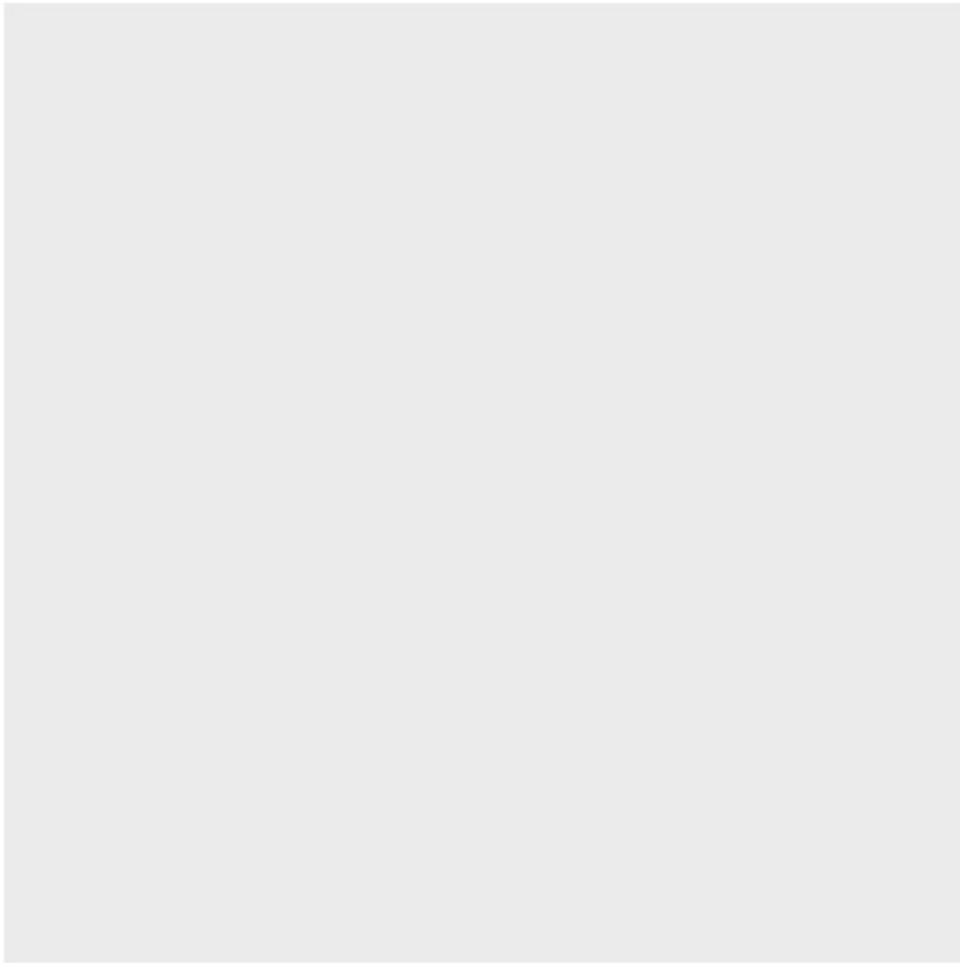
**Figure 9.1** An empty `ggplot()` call — no data was provided, so there's nothing to plot.

This is the default for `ggplot()`. Let's add some data (see figure 9.2).

```
ggplot(data = mtcars) ①
```

①

Once you're more familiar with the arguments, we'll stop explicitly naming them.



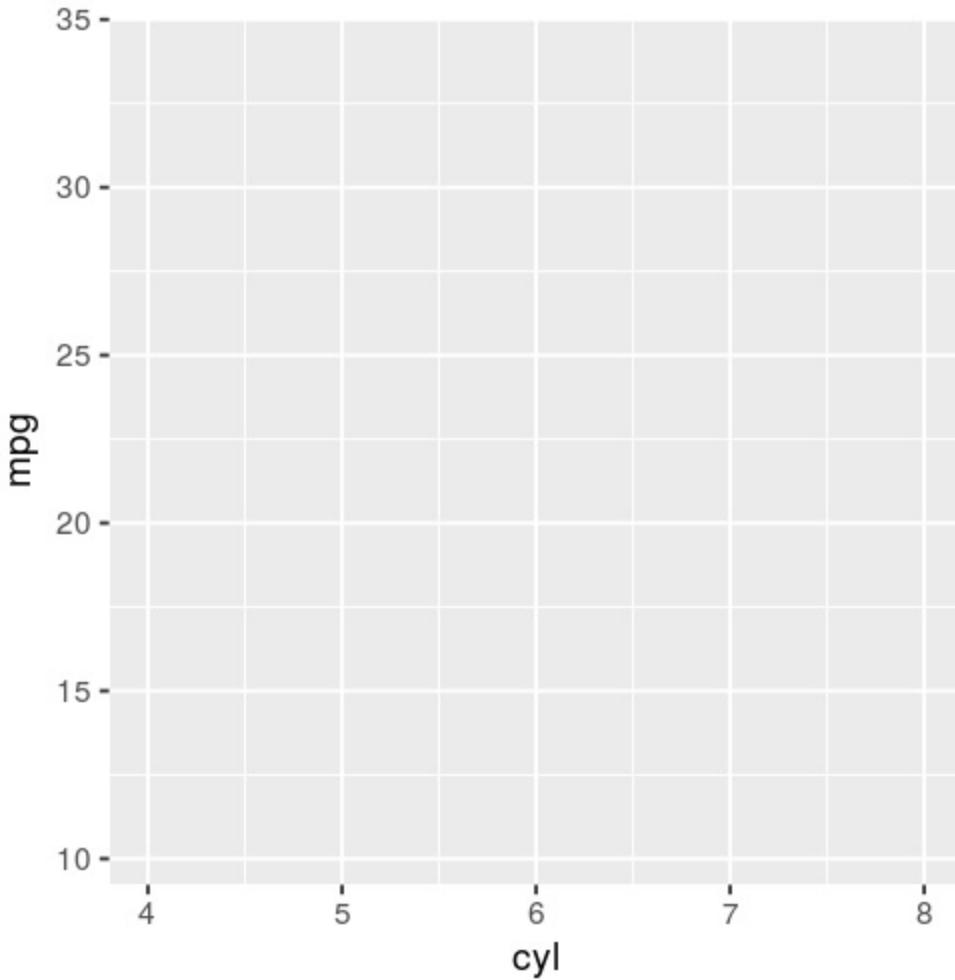
**Figure 9.2 You added some data, but there's still nothing to see.**

Nothing appears to have changed. `ggplot()` knows about the data, but not what you intend to *map* from the data to the plot. To do that, you specify the mapping using the `aes()` function (for *aesthetic*). It takes bare column names and generates the relevant mapping.<sup>1</sup> To plot the mileage (`mpg`) on the y-axis, and the number of cylinders (`cyl`) on the x-axis, you can provide these using `aes()` as the `mapping` argument (see [figure 9.3](#)):

---

<sup>1</sup> `aes()` also performs some validation on the values provided, such as autocompleting and autocorrecting potential aesthetic argument names.

```
ggplot(data = mtcars, mapping = aes(x = cyl, y = mpg))
```



**Figure 9.3 Adding a mapping using the aesthetic function `aes()` — the axes are now well defined, but you haven't specified what you want plotted.**

Now something is different: the axes have been titled with the relevant variable name, and the axes themselves have been arranged and labelled according to the values in `cyl` and `mpg`. Note that since `cyl` was provided as it appears in the `mtcars` object (a numeric vector that only has values 4, 6, and 8), the default presentation of the x-axis is to allow for any value in this range, with some automation of how to select the major and minor grid lines. If, instead, you want to treat this as a category (which it really is) then you can modify the aesthetic in place (figure 9.4):

```
ggplot(data = mtcars, mapping = aes(x = factor(cyl), y =
```

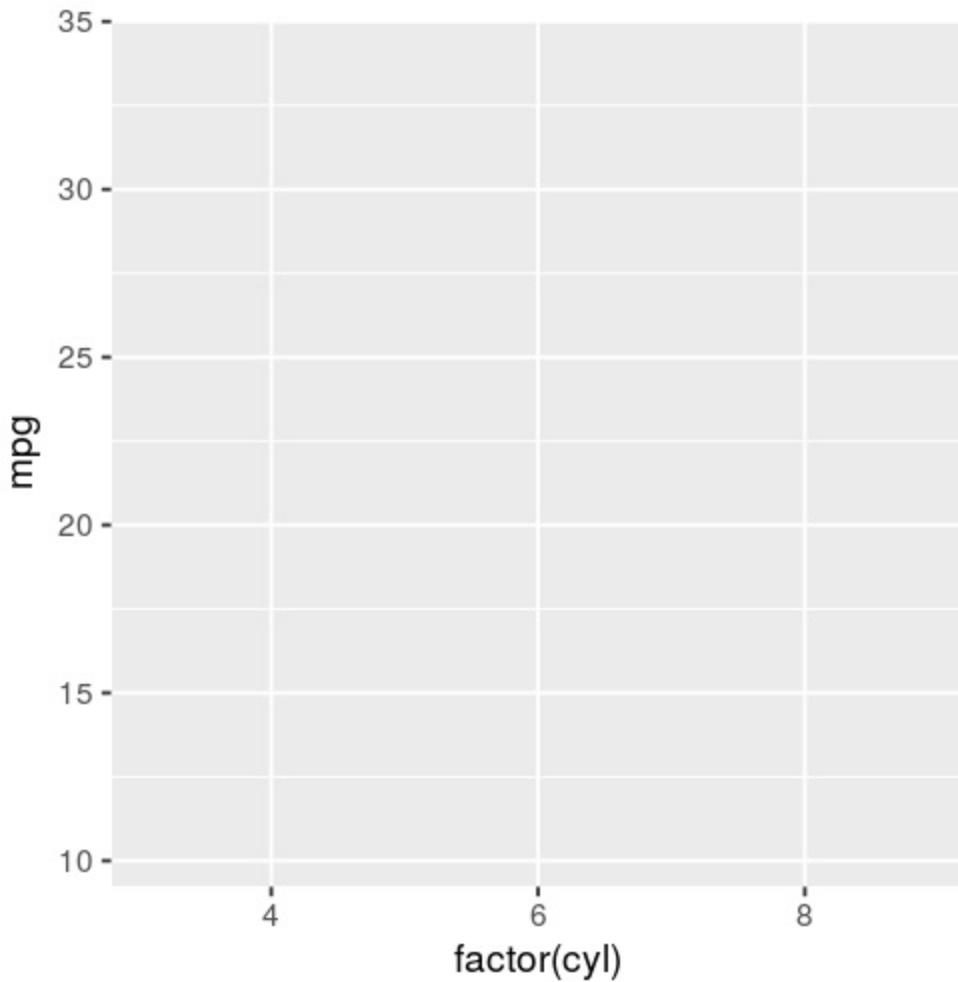
```
mpg))
```

Now the grid lines align with the only three possible values. Note also that the title for the x-axis reflects what you have provided within `aes()`. You'll see how to override this later.

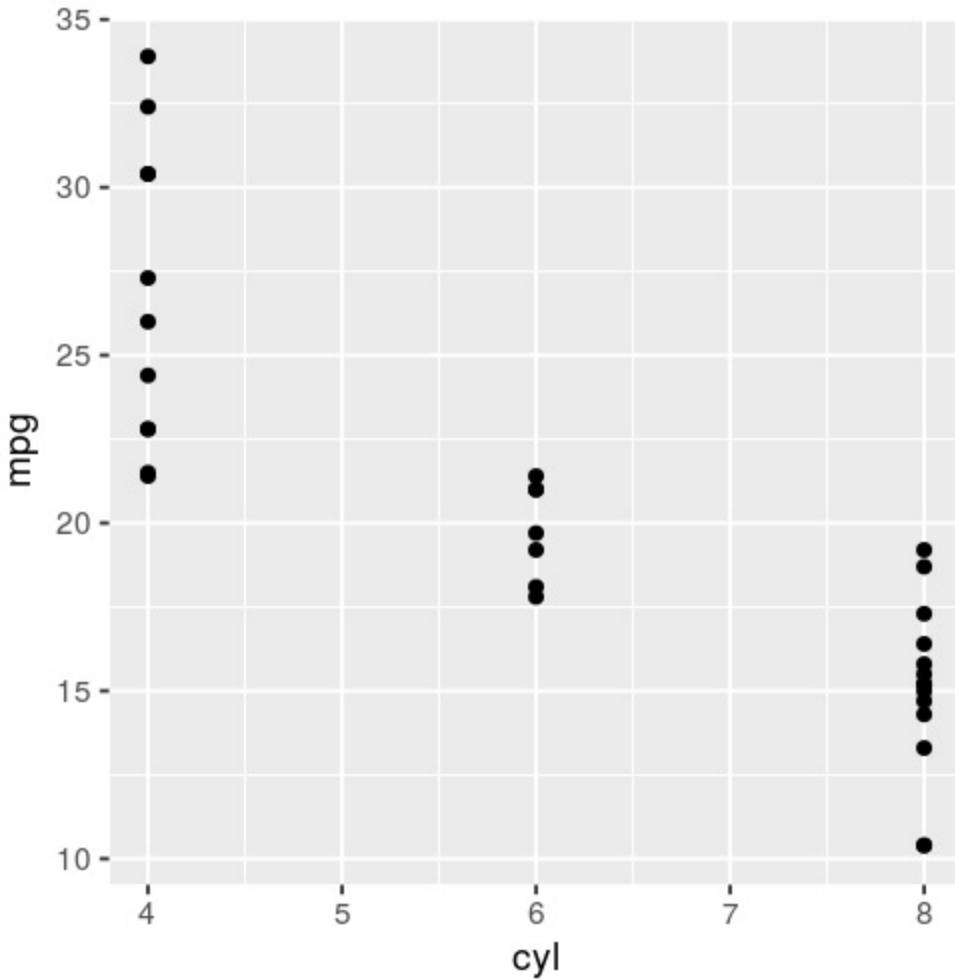
You still don't have any paired values on the plot. That's because you haven't specified the *geometry* you want. Each geometry begins with the prefix `geom_`; you can plot points with `geom_point()` (figure 9.5):

```
ggplot(data = mtcars, mapping = aes(x = cyl, y = mpg)) +  
  geom_point()
```

**NOTE** One thing to keep in mind is that `ggplot2` has a pen-and-paper model of constructing a plot; you begin with an empty canvas and add subsequent elements to it one or more at a time. These are added using `+`, which `ggplot2` extends such that it behaves appropriately when it encounters `ggplot2`'s class system (objects of class `gg`, `ggplot`, and related). Plots are stored just like anything else — as objects. This means that they can be saved (for example, with `saveRDS()`), recalled, and modified. One limitation due to the pen-and-paper model, though, is that elements can't be removed once the object is generated. You can certainly remove their addition from the code, but you can't remove them from a generated object (easily).



**Figure 9.4** Using `factor()` in the aesthetic, you can change how the axis is constructed.

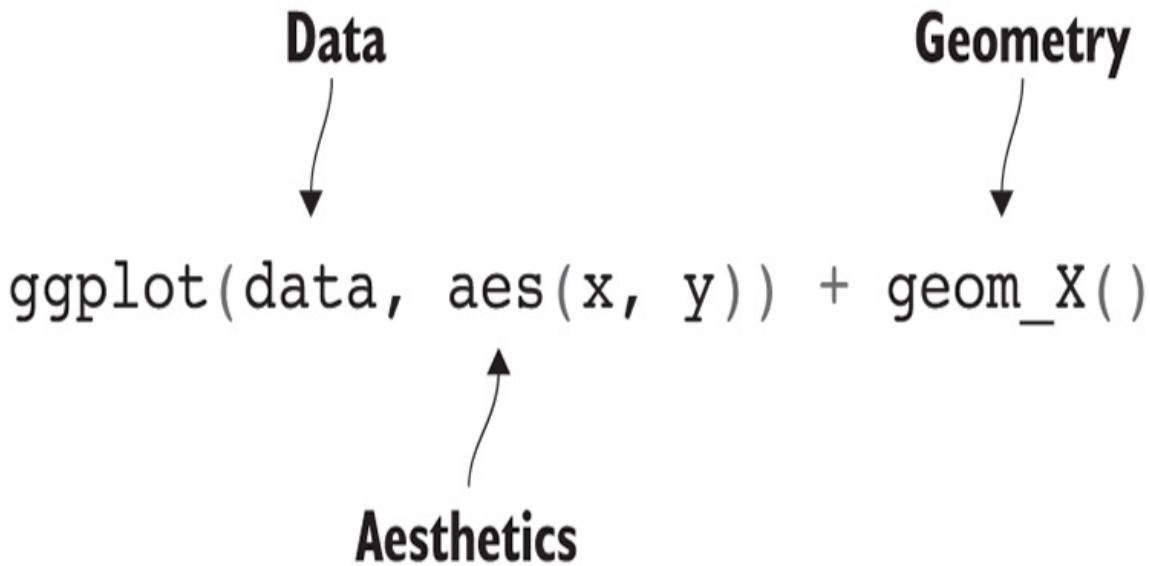


**Figure 9.5 Adding points as a geometry — with data, a mapping, and a geometry, you finally have some data plotted.**

Once you're familiar with the syntax, it's much cleaner to drop any unnecessary variable names and wrap calls in a pipe-like manner, which produces the same result as before:

```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point()
```

The general construction of a `ggplot()` call typically follows the pattern shown in [figure 9.6](#).

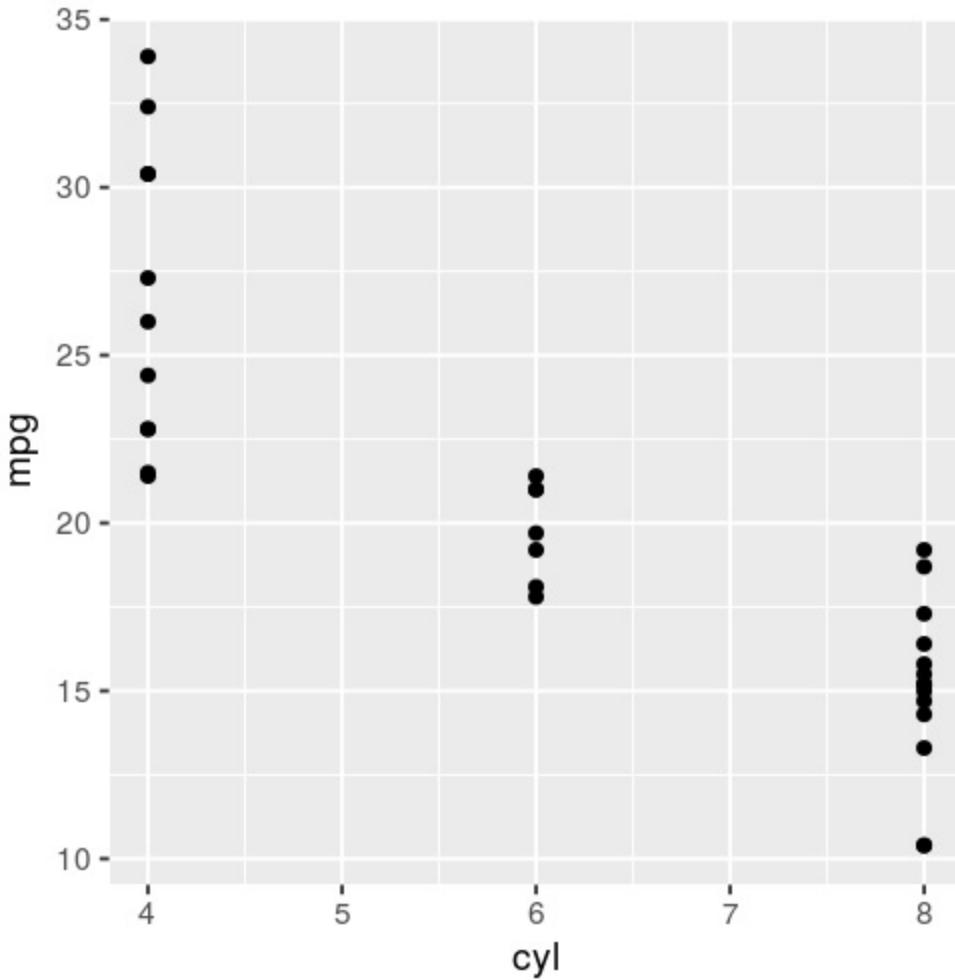


**Figure 9.6** `ggplot2` call construction

### 9.2.2 ADDING POINTS

You've already added some points to the plot with a simple `geom_point()` call. This took advantage of the fact that the `ggplot()` call included data and the aesthetics you wanted `geom_point()` to use. Alternatively, you can specify these within the `geom_point()` call (figure 9.7):

```
ggplot() + geom_point(data = mtcars, aes(cyl, mpg))
```



**Figure 9.7 Using aes() in geom\_point()**

Note that you've specified the `data` argument explicitly; that's because the arguments for `geom_point()` are, in order, mapping and *then* data. This means that failing to provide any argument names implies you want to use the data as the aesthetic. Fortunately, the error message suitably reminds you of this fact:

```
ggplot() + geom_point(mtcars, aes(cyl, mpg))

#> Error: ggplot2 doesn't know how to deal with data of
#> class uneval
```

Using the `data` and `aes` arguments in the `ggplot()` call has an advantage: the subsequent `geom_` calls inherit these values, so you can add several with the same arguments passed on. But any new arguments are only local to that call ([figure 9.8](#)):

```
ggplot(mtcars, aes(cyl, mpg, shape = factor(am))) + ①  
  geom_point(size = 8, col = "lightblue") + ② ③  
  geom_point() ④
```

①

The shape aesthetic will be inherited by all geoms unless overridden.

②

Produces large, light blue points

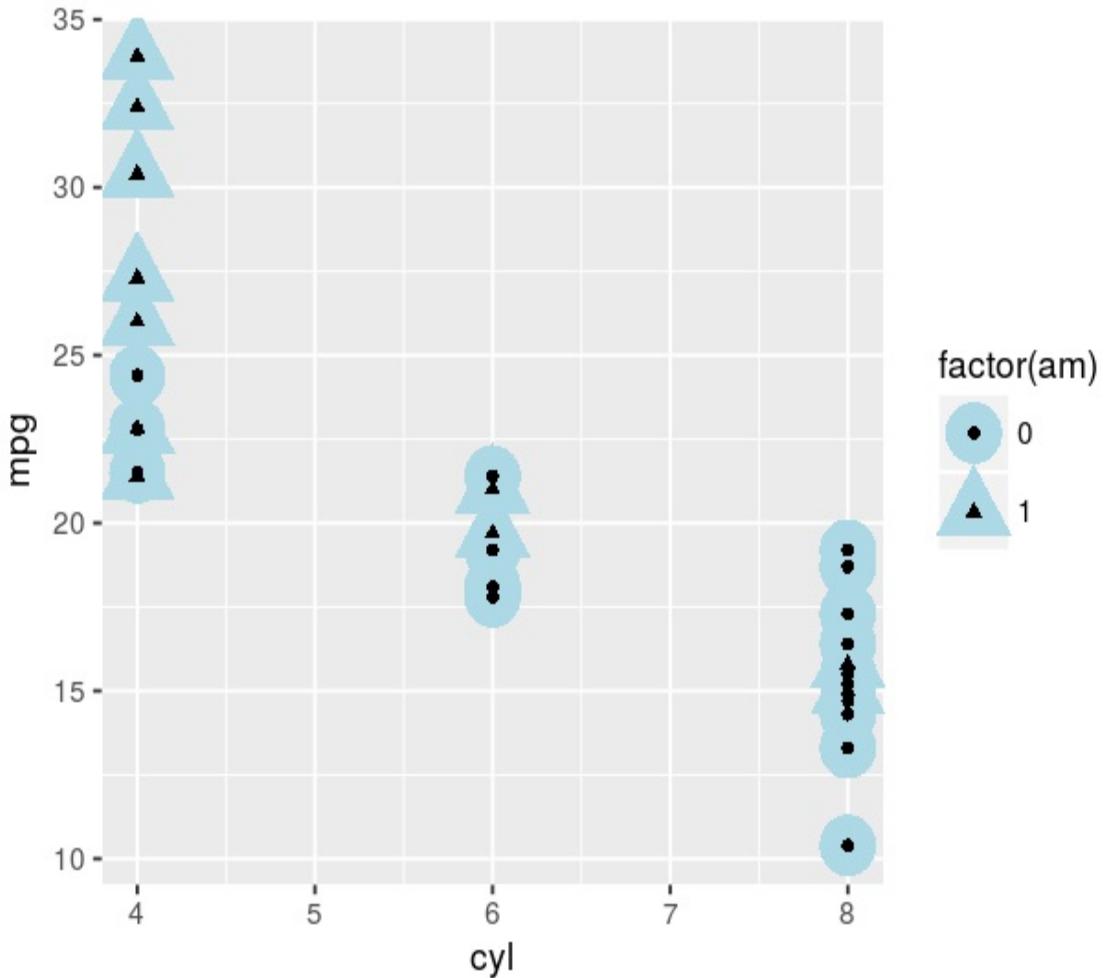
③

Note that you don't specify the shape in the geom\_ calls — these are inherited.

④

Produces the default points, on top of the earlier points

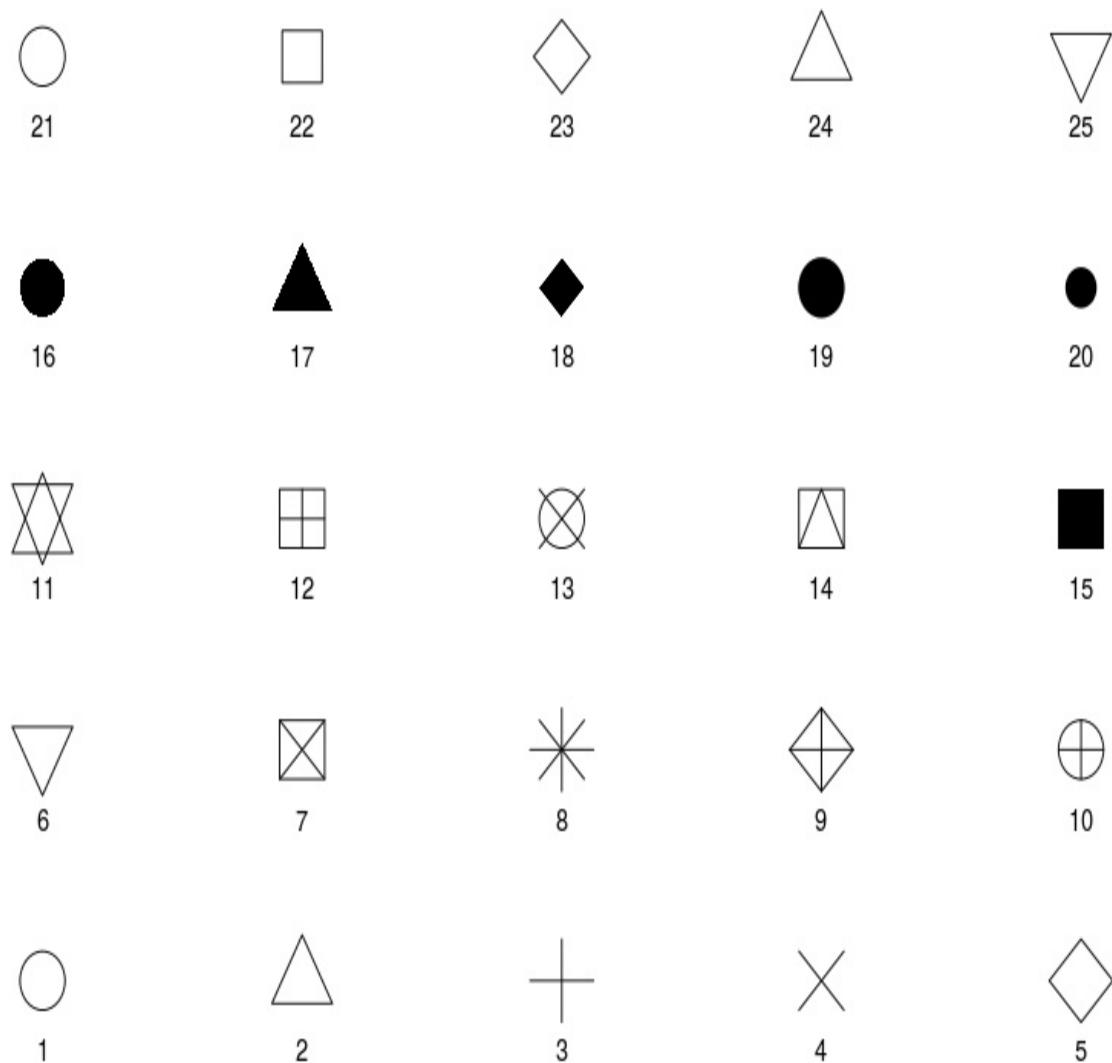
**NOTE TO PRINT BOOK READERS** Many graphics in this book are meant to be viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to [www.manning.com/freebook](http://www.manning.com/freebook) and follow the instructions to complete your pBook registration.



**Figure 9.8** Aesthetics can be inherited by geometries if they're provided in the `ggplot()` call.

Because `ggplot()` follows the pen-and-paper model of plotting, the order in which you request the `geom_`s determines the order in which they are drawn. If you have features you want to appear above or below other features, then change the order in which they are added appropriately.

The point shapes that can be plotted , alternatively denoted by `pch`, are numbered 1 to 25 and are shown in figure 9.9. The default point shape is 16, which produces a filled circle.



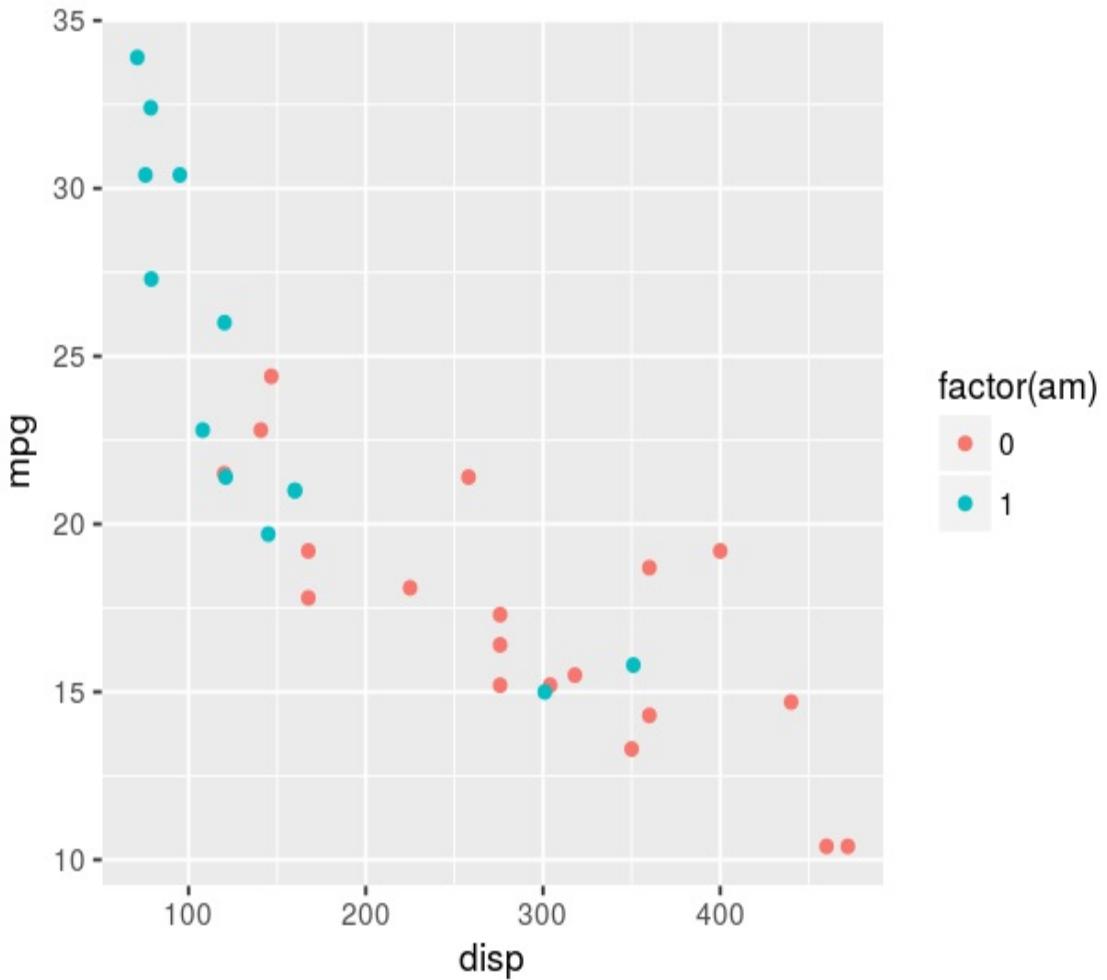
**Figure 9.9** Plotting points with `pch` values. The default point shape is number 16.

### 9.2.3 STYLE AESTHETICS

The point symbol shape in the previous example is an example of styling. The styling can be added either to individual `geom_` calls for all members of that `geom_` (such as the `size` and `col` styling earlier); to individual `geom_` calls grouped by some quantity (within `aes()`, such as the shape styling earlier); or to all compatible `geom_` calls as an aesthetic (within the `ggplot()` `aes()` call, to be inherited by the `geom_s`). This allows a great deal of flexibility in how you construct illustrative plots.

To make clearer the distinction between manual and automatic transmission vehicles when visualizing the `mpg` at values of `disp` in the `mtcars` dataset, you can add color (distinguished by the value of `am`, the transmission column — see figure 9.10):

```
ggplot(mtcars, aes(disp, mpg, col = factor(am))) +
  geom_point()
```



**Figure 9.10** Using color as an aesthetic via `aes()`

Again, you used `factor(am)` to let `ggplot()` know that you want to treat these as categories, despite them currently being numbers.

You can have many colors in the grouping: `ggplot()` will automatically select them from a suitable set of predefined colors.<sup>2</sup> If you have three groups, the colors are slightly different

than before, as shown in figure 9.11:

---

<sup>2</sup> In the current version of `ggplot2`, these are chosen to be equally spaced hues around a color wheel, starting at 15 degrees.

---

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point()
```

Choosing a specific color to use can be quite tricky, as there are many named colors from which to select. The full list of named colors can be found using `colors()` (or `colours()`), a sample of which is shown here (see figure 9.12):

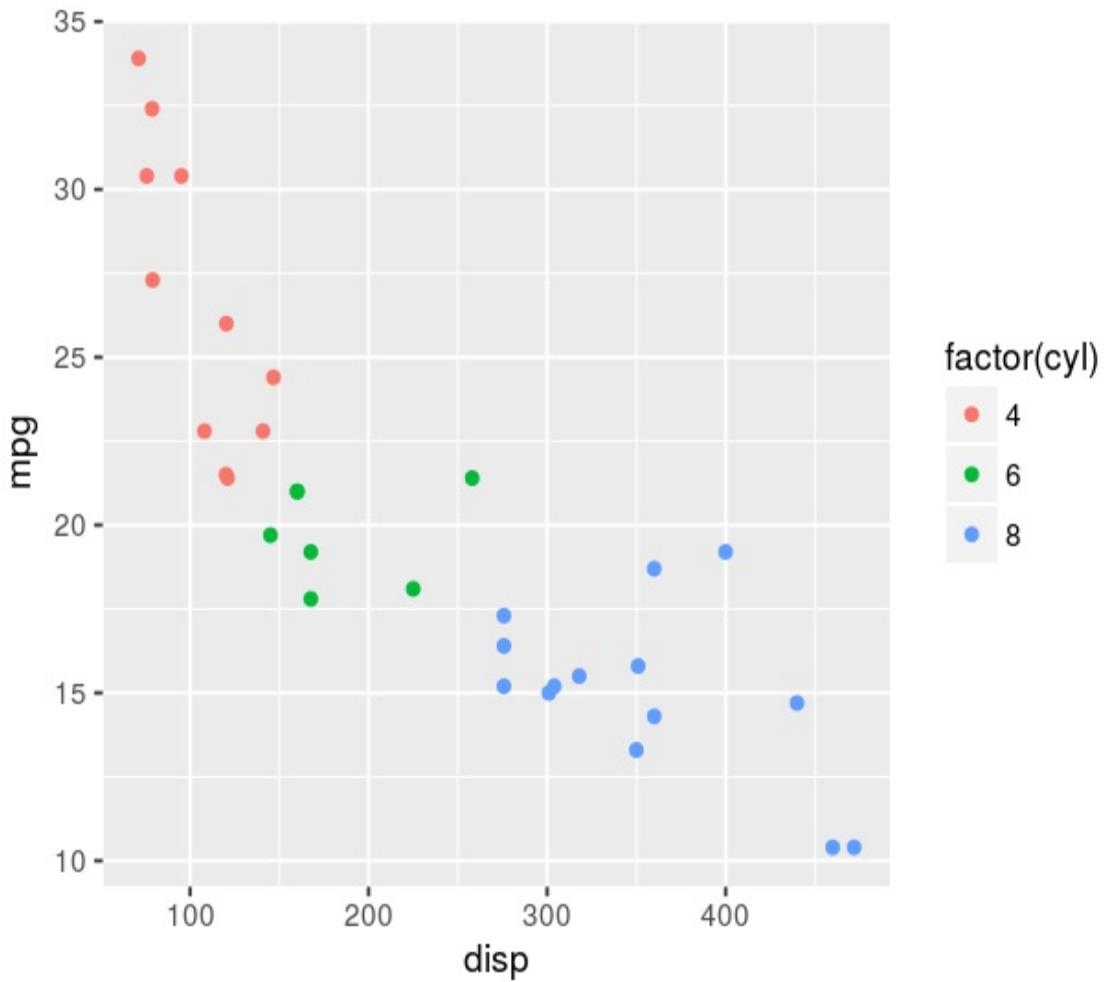
```
set.seed(13)      ①
sample(colors(), 10)    ②
#> [1] "mediumpurple" "gray9"       "green2"
"coral3"
#> [5] "thistle4"     "aquamarine"   "indianred2"
"olivedrab4"
#> [9] "saddlebrown"   "blue1"
```

①

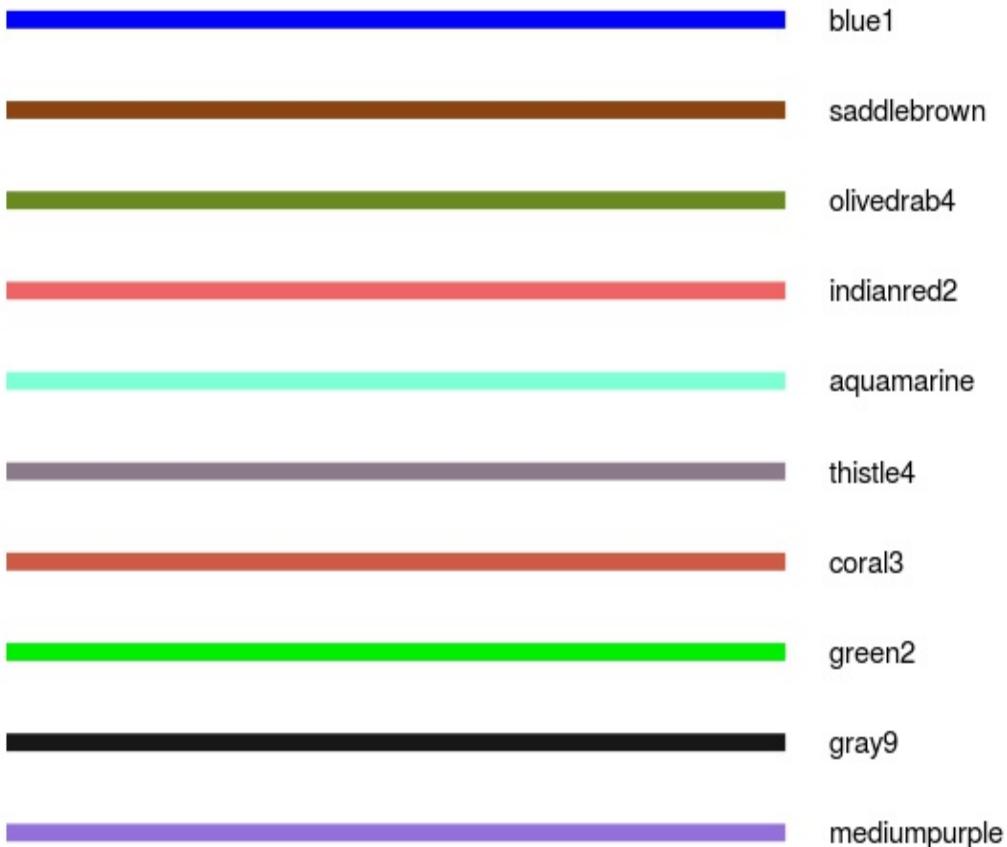
This seed was chosen because the random selection shows a wide variety of the available colors.

②

**sample() takes a random selection from a vector without replacement. If you want random selection with replacement, set replace = TRUE.**



**Figure 9.11** Note the cyl vs. disp relationship that becomes apparent.



**Figure 9.12 Examples of colors available to `ggplot()`**

Any color at all can be used if you specify either its RGB combination using `rgb()`,<sup>3</sup> or many of the other specifications for colors. Reading through `?colors` is a good place to start.

---

<sup>3</sup> Red, green, blue, in the range 0-255.

The complete list of styles and aesthetics that can be applied is available with a non-exported (not intended for use within your code) function:

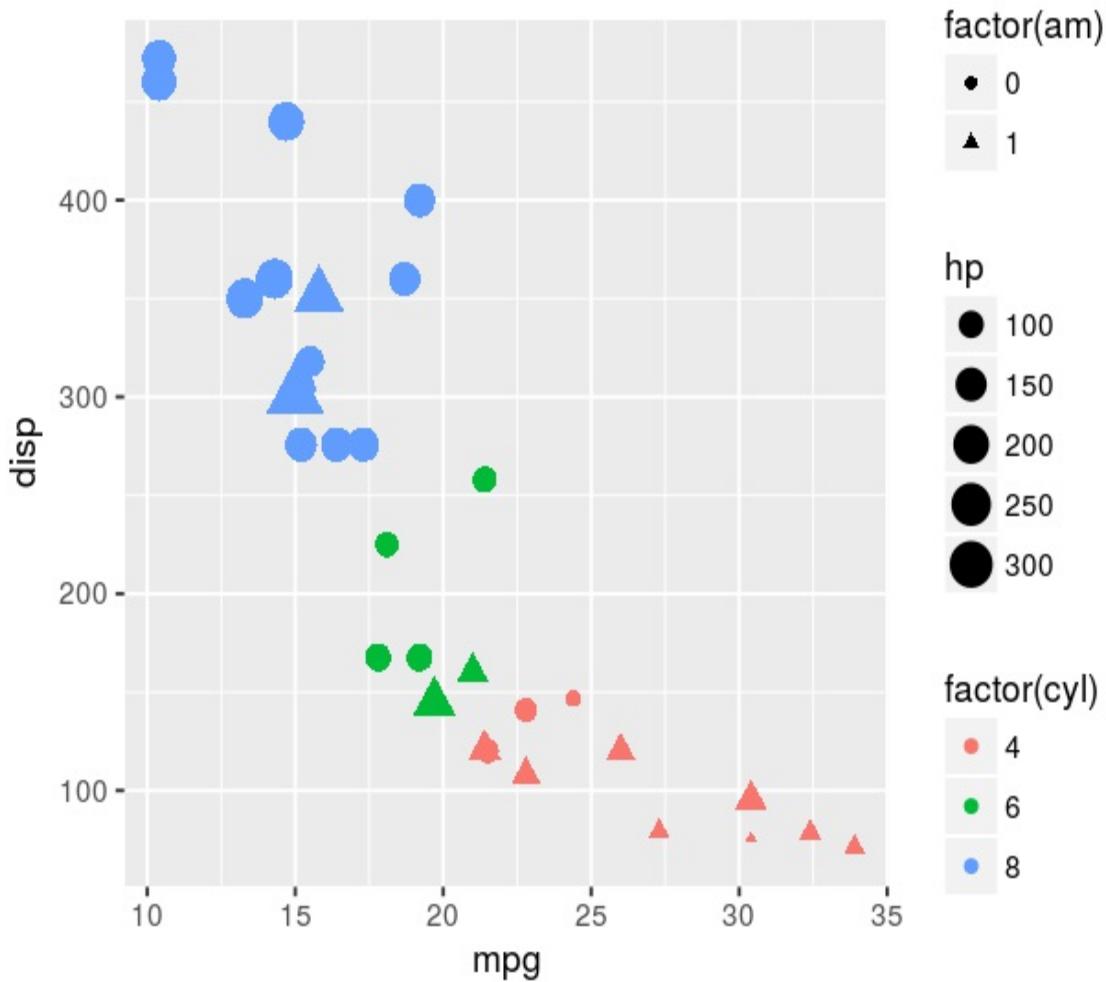
```
ggplot2:::all_aesthetics
#> [1] "adj"        "alpha"       "angle"       "bg"
"cex"
#> [6] "col"        "color"       "color"       "fg"
"fill"
#> [11] "group"     "hjust"      "label"      "linetype"
"lower"
#> [16] "lty"        "lwd"        "max"        "middle"
```

```
"min"  
#> [21] "pch"          "radius"        "sample"        "shape"  
"size"  
#> [26] "srt"          "upper"         "vjust"         "weight"  
"width"  
#> [31] "x"            "xend"         "xmax"         "xmin"  
"xintercept"  
#> [36] "y"            "yend"         "ymax"         "ymin"  
"yintercept"  
#> [41] "z"
```

Many of these are straightforward enough to guess at their usage. Not every aesthetic can be applied to every `geom_`, but the most commonly used ones — changing color, linetype, size, fill color, or transparency (`alpha`) — are generally available.

You can add as many of these as you want to help convey the information in your data as another dimension to the visualization (figure 9.13):

```
ggplot(data = mtcars,  
       mapping = aes(x = mpg,  
                      y = disp,  
                      col = factor(cyl),  
                      shape = factor(am),  
                      size = hp))  
+  
  geom_point()
```

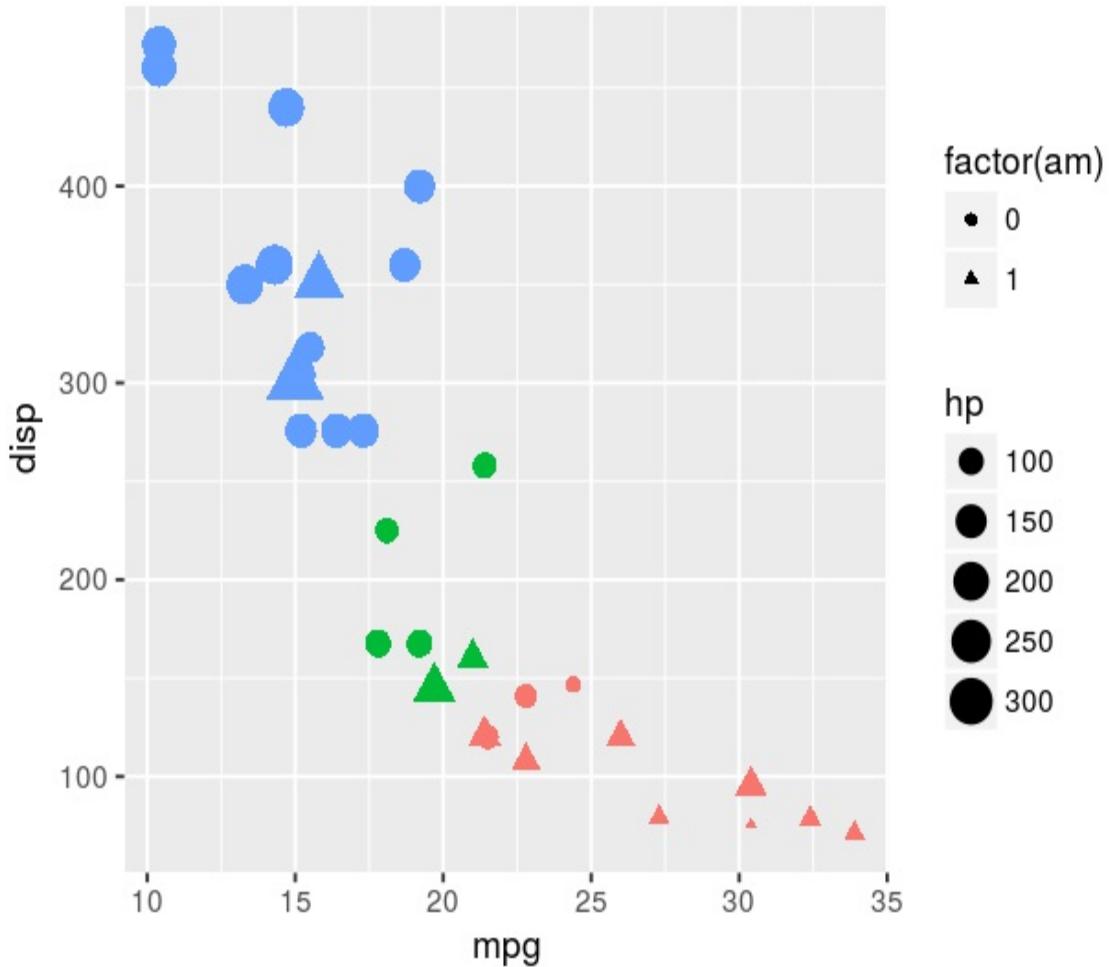


**Figure 9.13** More dimensions (as aesthetics) help to distinguish different aspects of the data and highlight relationships

When groupings are added via `aes()`, a *legend* is produced alongside the plot as a guide to which symbol/shape/color represents which group. These too can be adjusted — for example, removing the color legend, using `guides()` (figure 9.14):

```
ggplot(data = mtcars,
       mapping = aes(x = mpg,
                      y = disp,
                      col = factor(cyl),
                      shape = factor(am),
                      size = hp))
  +
  geom_point() +
  guides(color = FALSE) ①
```

(1)

**No color legend.****Figure 9.14 Removing a legend using guides()**

## 9.2.4 ADDING LINES

You can add one or more lines to the plot as a geometry, using `geom_line()`. For this example, you need some data more easily represented by a line than the `mtcars` values, so you can create some:

```
# install.packages("dplyr")
library(dplyr) ①
#
#> Attaching package: 'dplyr'
```

```
# The following objects are masked from 'package:stats':  
#>  
#>     filter, lag  
# The following objects are masked from 'package:base':  
#>  
#>     intersect, setdiff, setequal, union  
  
sin_df <- data.frame(theta = seq(pi, 0, -0.5)) %>%      ②  
  mutate(x = cos(theta), y = sin(theta))      ③
```

①

You can use what you've learned about dplyr to make this easier.

②

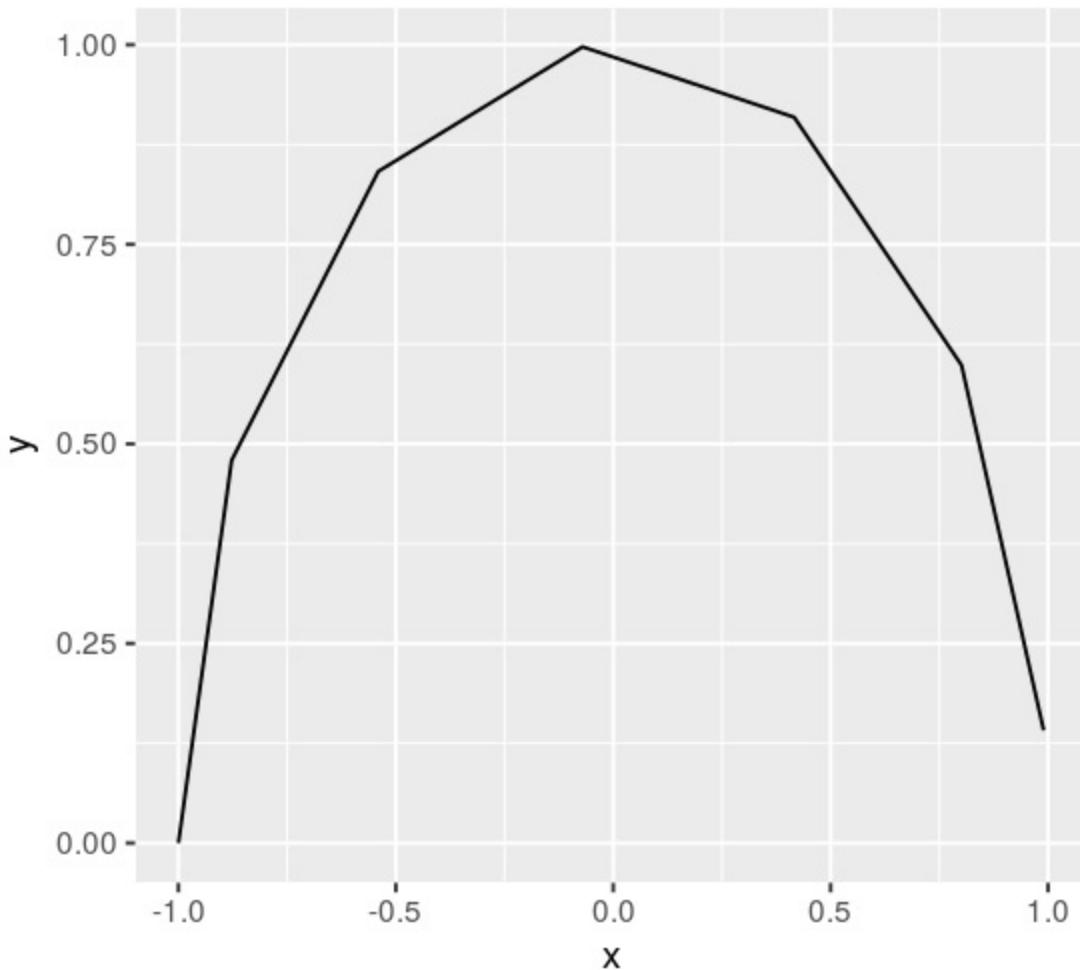
Starts with a simple data.frame containing values of an angle, theta

③

Takes advantage of dplyr's ability to reuse data to create two new columns derived from theta. The x and y values represent points on a circle.

This can be plotted as just lines joining the pairs of x and y values, as in figure 9.15:

```
ggplot(sin_df, aes(x, y)) + geom_line()
```



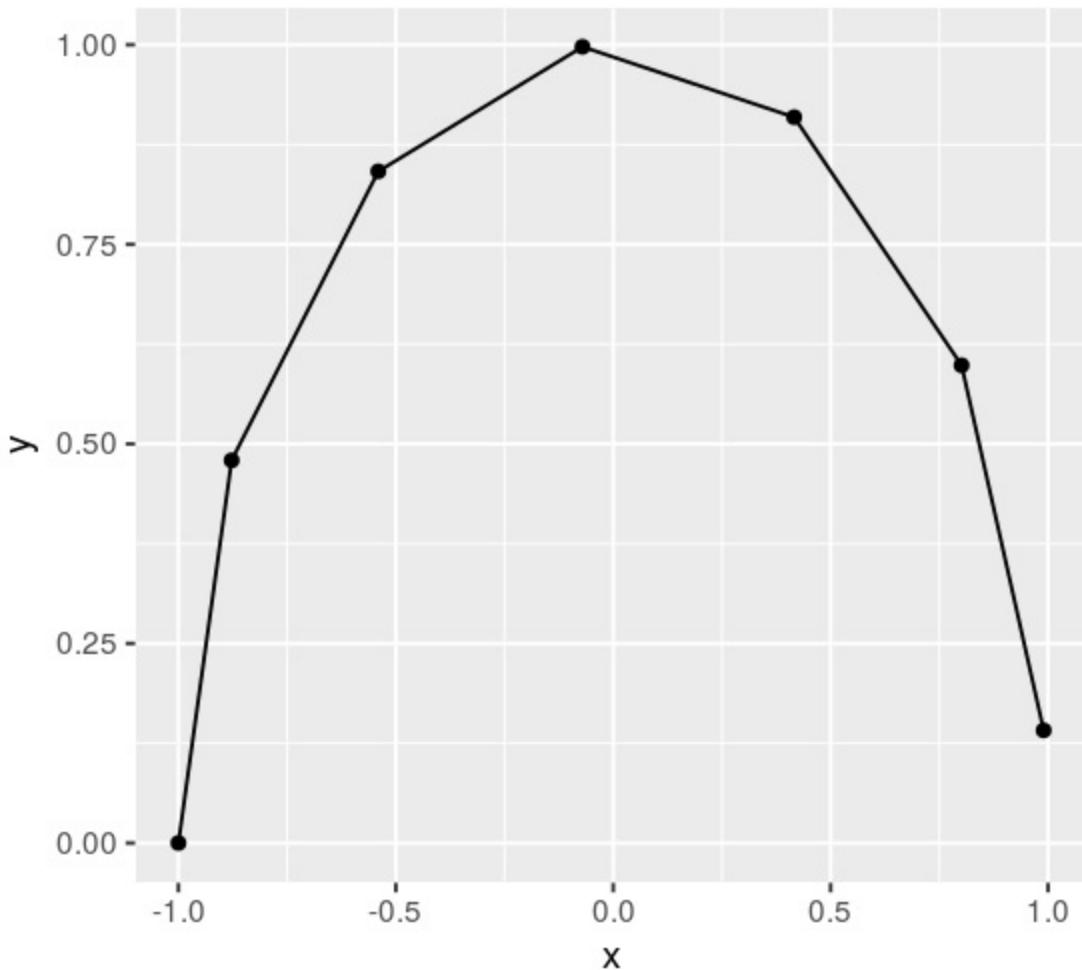
**Figure 9.15 Lines joining pairs of x and y values**

Or you can specifically add the points ([figure 9.16](#)):

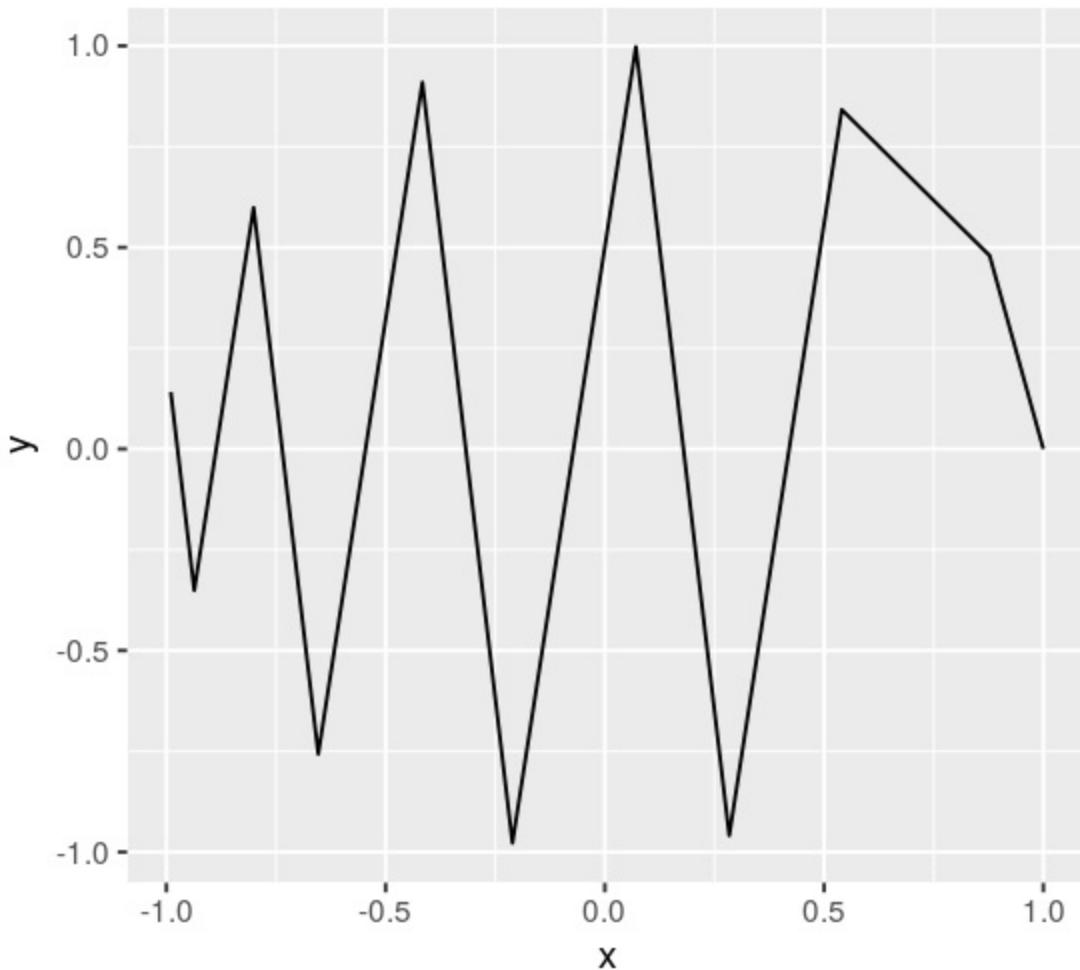
```
ggplot(sin_df, aes(x, y)) + geom_line() + geom_point()
```

If you construct the data to cover more of a circle, you run into the problem that `ggplot()` tries to join up the paired values in a way you perhaps don't want ([figure 9.17](#)):

```
sin_df <- data.frame(theta = seq(0, 1.75*pi, 0.5)) %>%
  mutate(x = cos(theta), y = sin(theta))
ggplot(sin_df, aes(x, y)) + geom_line()
```



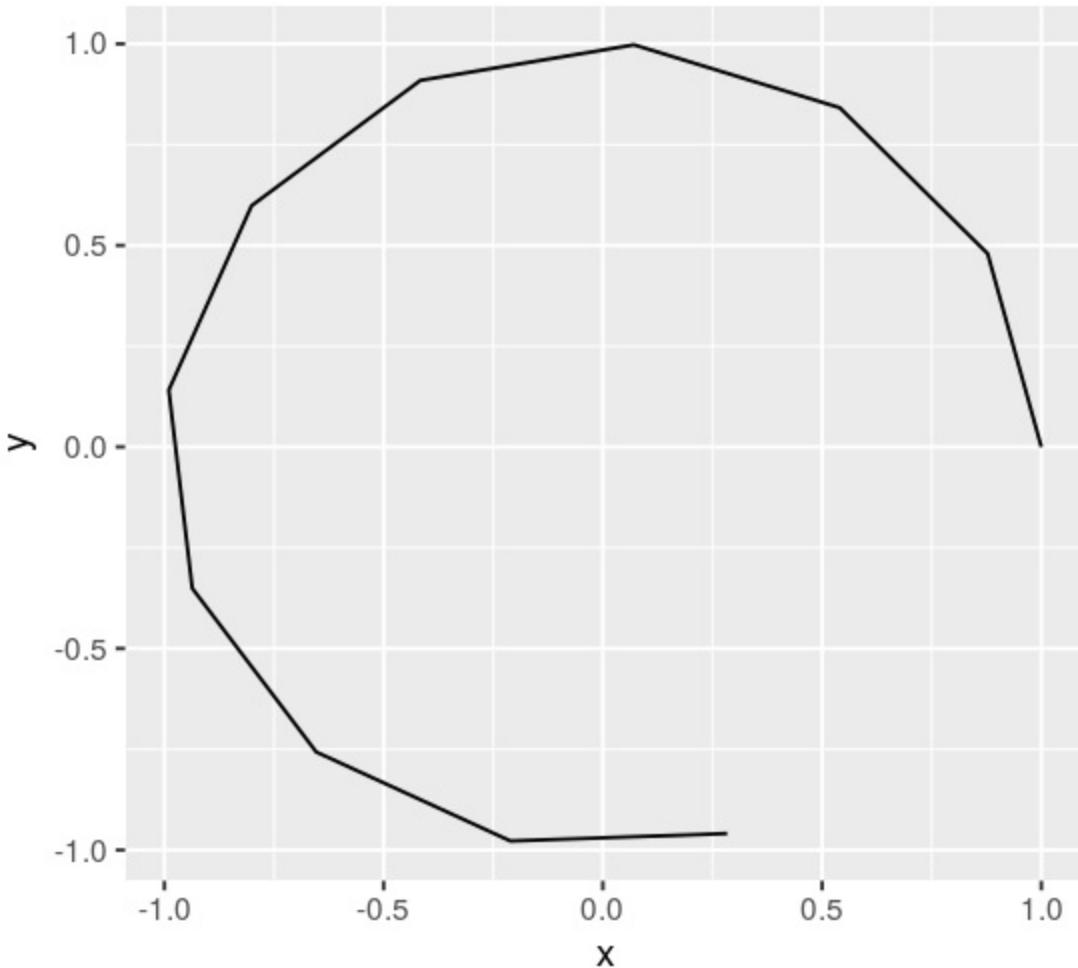
**Figure 9.16 Lines and points joining pairs of values**



**Figure 9.17 This probably isn't what you want.**

The reason for this behavior is that the points are joined as if they were sequentially produced along the x-axis, which isn't the case for a circle. In this case, you can use an alternative: `geom_path()` instead assumes the paired points are sequential, as in [figure 9.18](#):

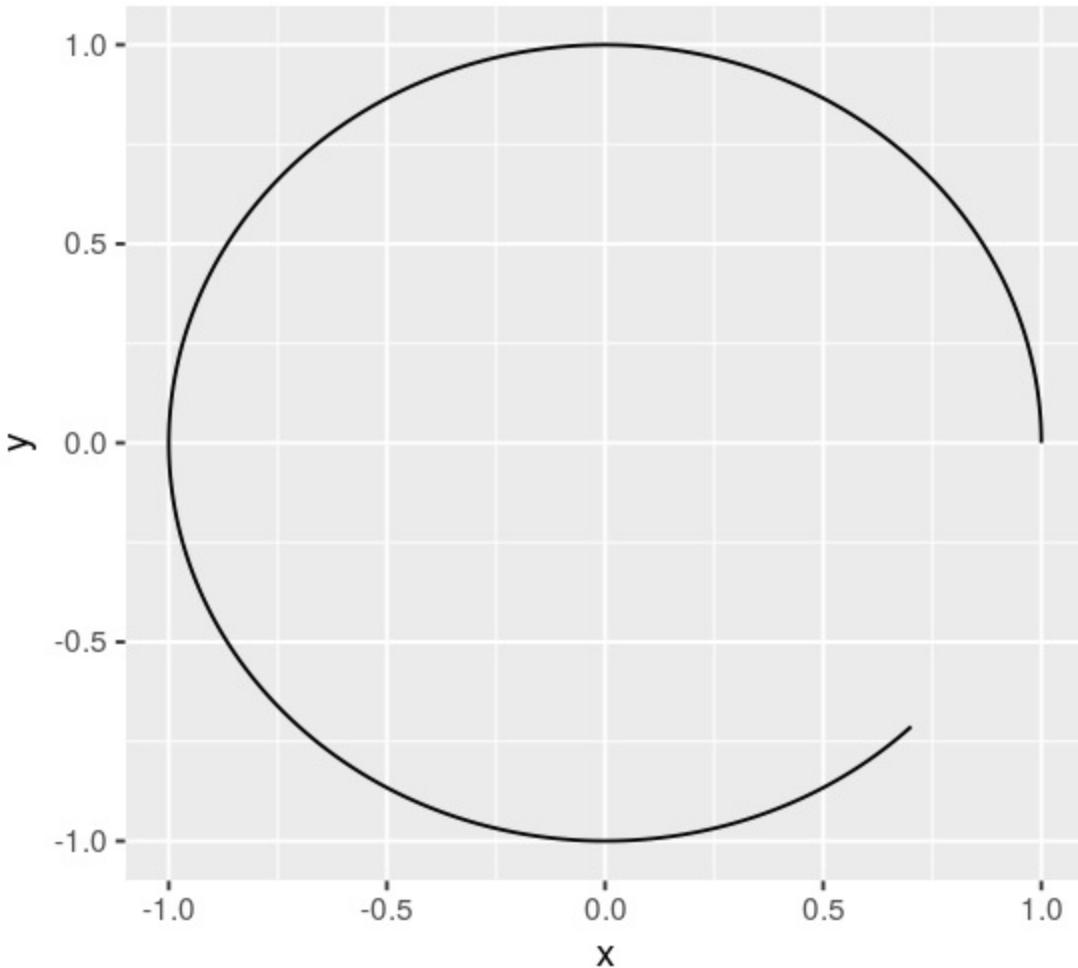
```
ggplot(sin_df, aes(x, y)) + geom_path()
```



**Figure 9.18** Plotting the points as a path assumes the pairs are to be plotted sequentially rather than in the order of the  $x$  values.

If you make the data more finely scaled (with the `by` argument to `seq()`), you can make the plot look more smoothly circular, as in [figure 9.19](#):

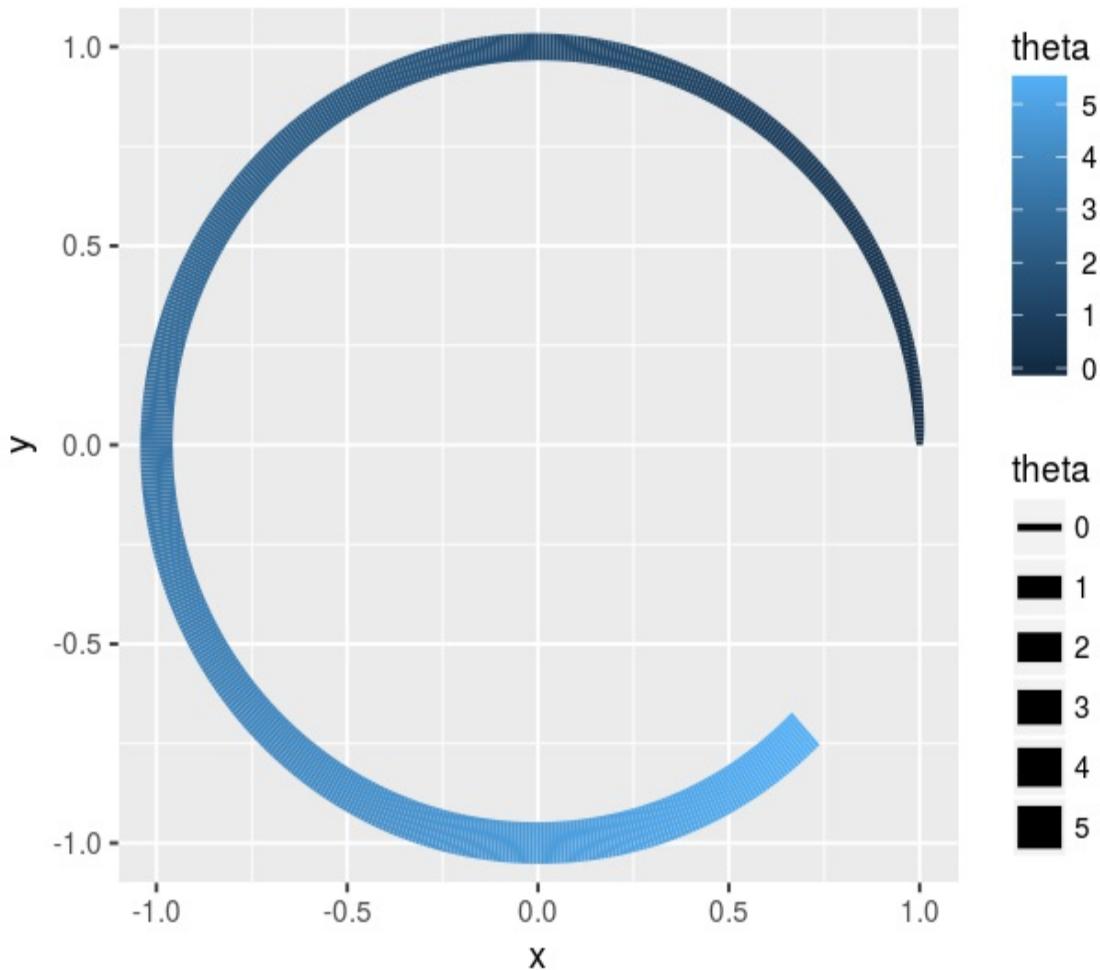
```
sin_df <- data.frame(theta = seq(0, 1.75*pi, 0.01)) %>%
  mutate(x = cos(theta), y = sin(theta))
ggplot(sin_df, aes(x, y)) + geom_path()
```



**Figure 9.19** A more finely scaled data object produces a smoother curve when joining the points by lines.

You can apply some differential aesthetics to the path segments, such as line thickness (`lwd`) and color (`col`), as in [figure 9.20](#):

```
ggplot(sin_df, aes(x, y)) + geom_path(aes(lwd = theta, col = theta))
```



**Figure 9.20** Lines can be styled just as points can.

Note here that you *are* now taking advantage of the fact that theta isn't a factor, and the color scale smoothly transitions from one color to another.

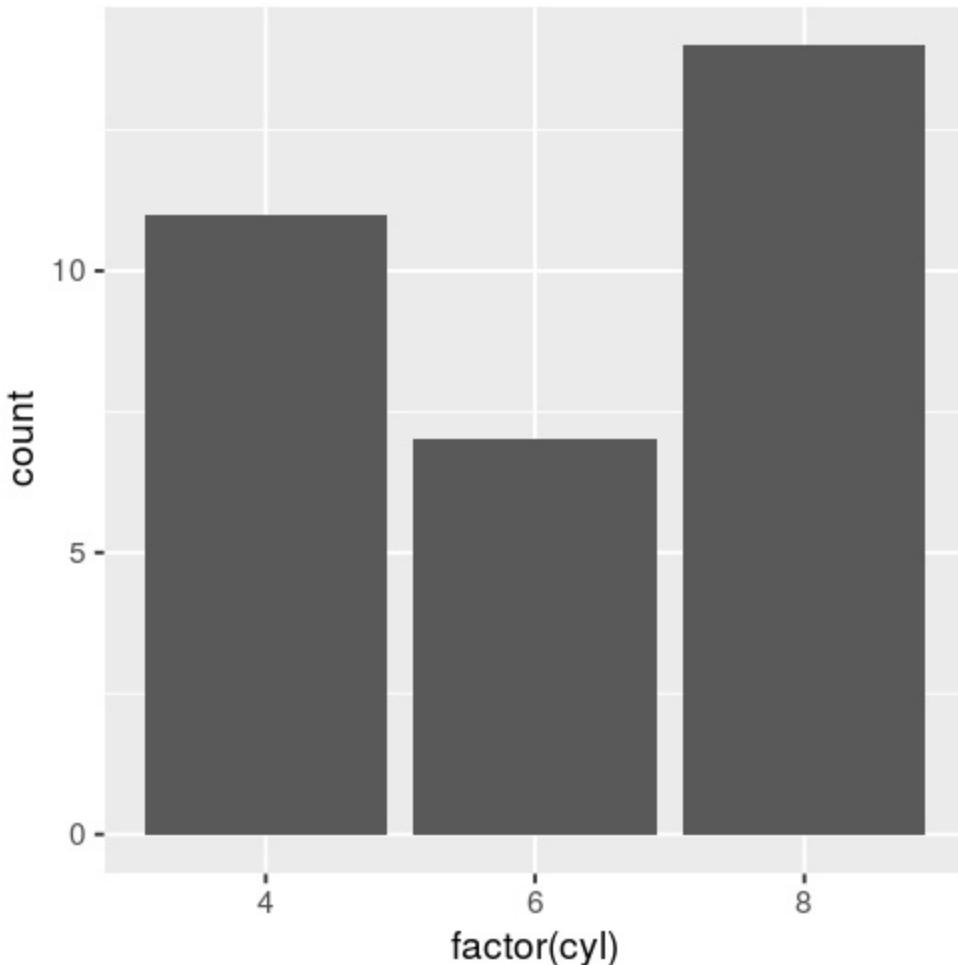
## 9.2.5 ADDING BARS

When you have categorical data such as counts of things, it's common to plot them as a barplot. One option would be to do the counting of the things manually, but `ggplot2` is clever enough to do the work for you. To plot the number of vehicles in the `mtcars` dataset, which have certain values of `cyl` (number of cylinders), you can use `geom_bar()`, resulting in figure 9.21:

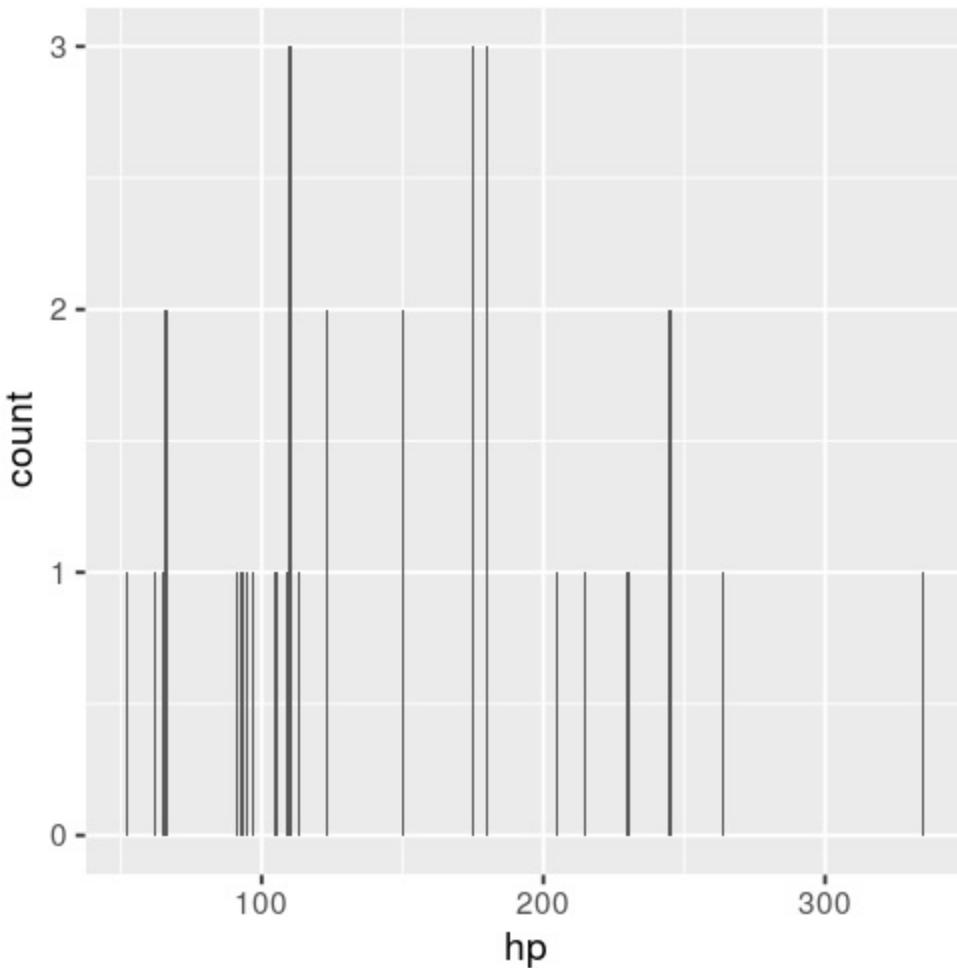
```
ggplot(mtcars) + geom_bar(aes(factor(cyl)))
```

This works for any variable, even if it isn't so coarsely categorized, and even if it's continuous, such as figure 9.22:

```
ggplot(mtcars) + geom_bar(aes(hp))
```



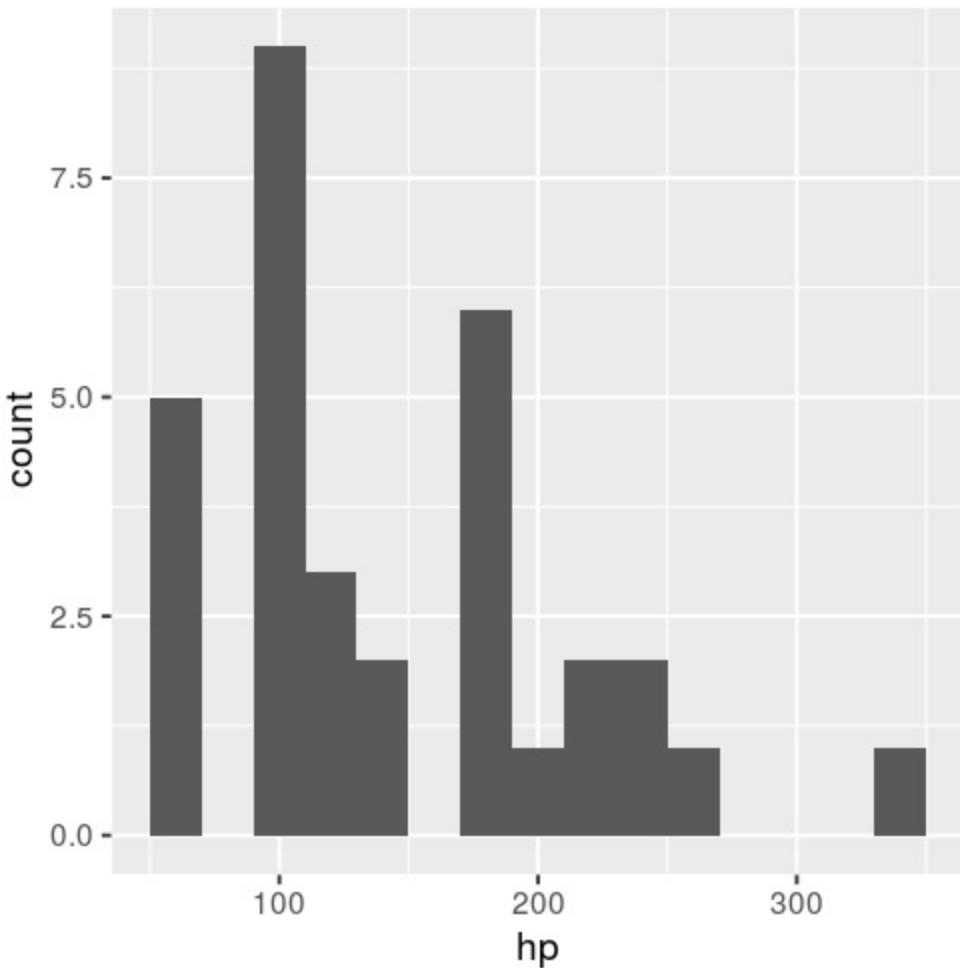
**Figure 9.21** Categorical values can be counted and plotted as a barplot.



**Figure 9.22 Continuous values have counts in each distinct value.**

Here the counts of entries with a certain value of `hp` have been plotted as thin lines at those values on the x-axis. Note that you only need to specify the `x` argument; the count is generated within `geom_bar()`. Alternatively, you may consider these as individual groups with some width. If you group these `hp` values into bins 20 units wide, you can produce a histogram, such as in figure 9.23:

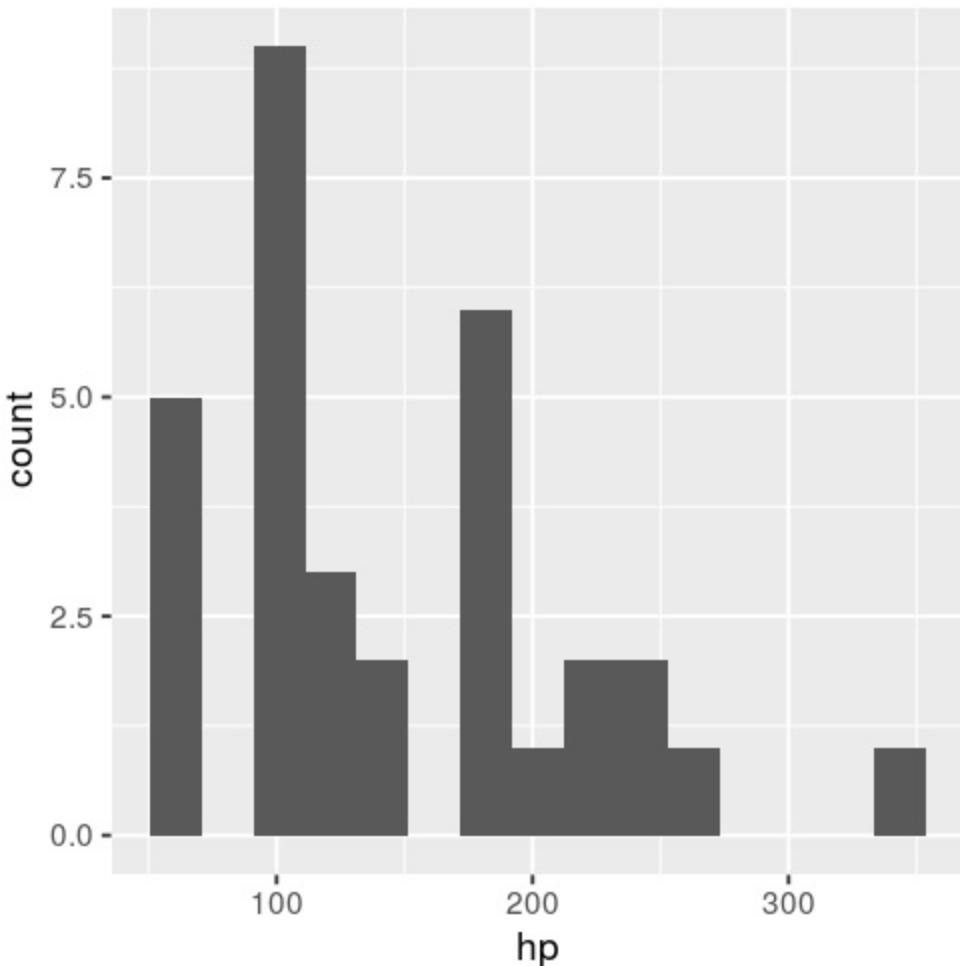
```
ggplot(mtcars, aes(hp)) + geom_histogram(binwidth = 20)
```



**Figure 9.23** A *histogram* is a barplot with some binning.

This looks similar to the `geom_bar(aes(hp))` plot but with wider (20 units) bars. Rather than the width, you can select the number of bins into which the values will be grouped, such as figure 9.24:

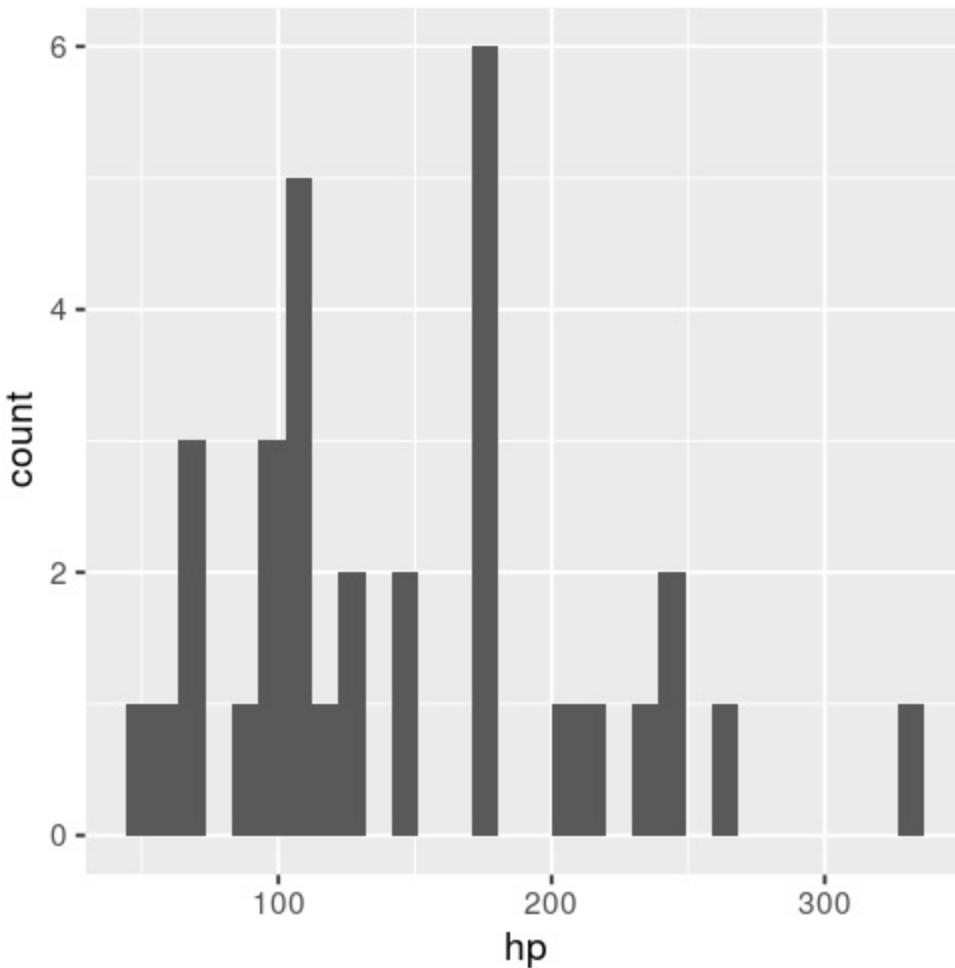
```
ggplot(mtcars, aes(hp)) + geom_histogram(bins = 15)
```



**Figure 9.24 Specifying 15 bins rather than the width of the bins**

The value of `bins` has a default of 30 (see figure 9.25), and this produces a message if you rely on it:

```
ggplot(mtcars, aes(hp)) + geom_histogram()  
# `stat_bin()` using `bins = 30`. Pick better value with  
`binwidth`.
```



**Figure 9.25** The default number of bins is 30.

Another common scenario is to specify the heights of the bars. Here you manually count the number of vehicles with cyl cylinders (using your dplyr skills), as well as the average miles per gallon (mpg) within those groups:

```
average_mpg_by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(n_cars = n(),             ①
            avg_mpg = mean(mpg))      ②
  )                                ③
```

①

Groups by number of cylinders, cyl

(2)

**Calculates the number of vehicles in each group, n\_cars, with the helper function n(), which counts records within a group**

(3)

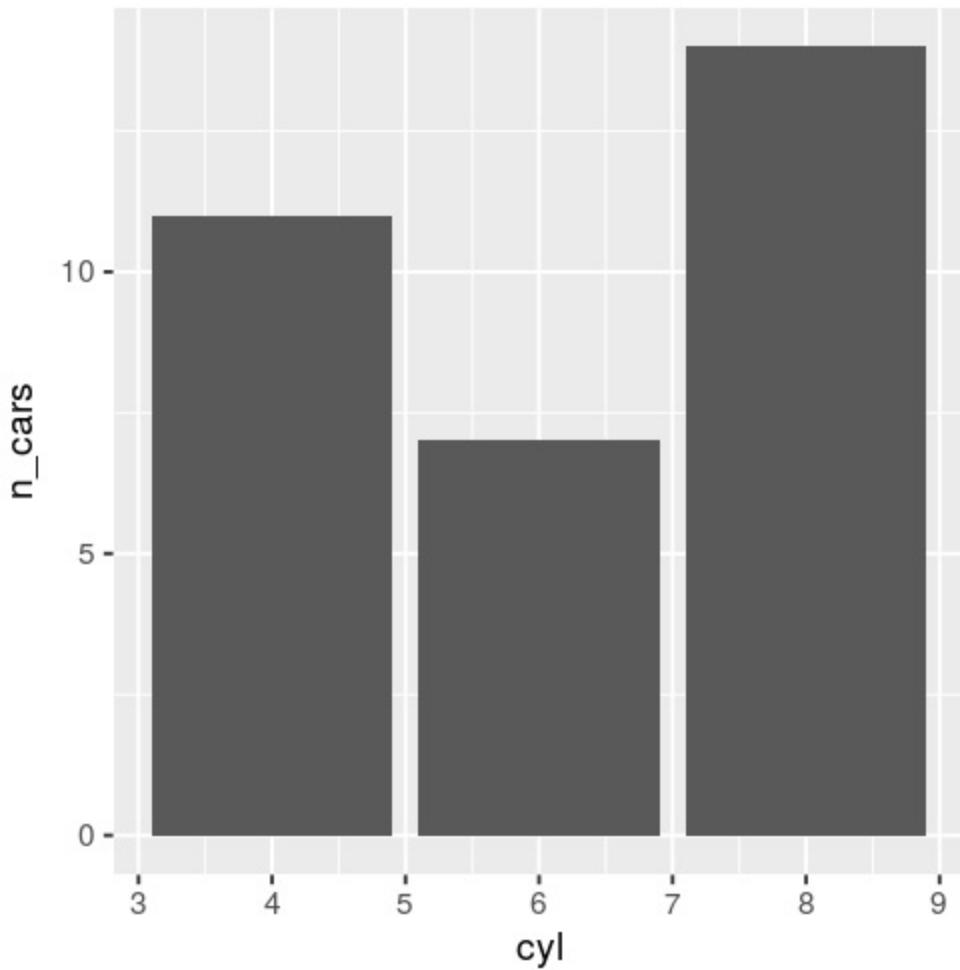
**Calculates the average miles per gallon in each group, avg\_mpg**

You can reproduce the preceding `geom_bar()` result manually (see figure 9.26) by specifying that the statistic to run over this data is the `identity` transformation (no transformation):

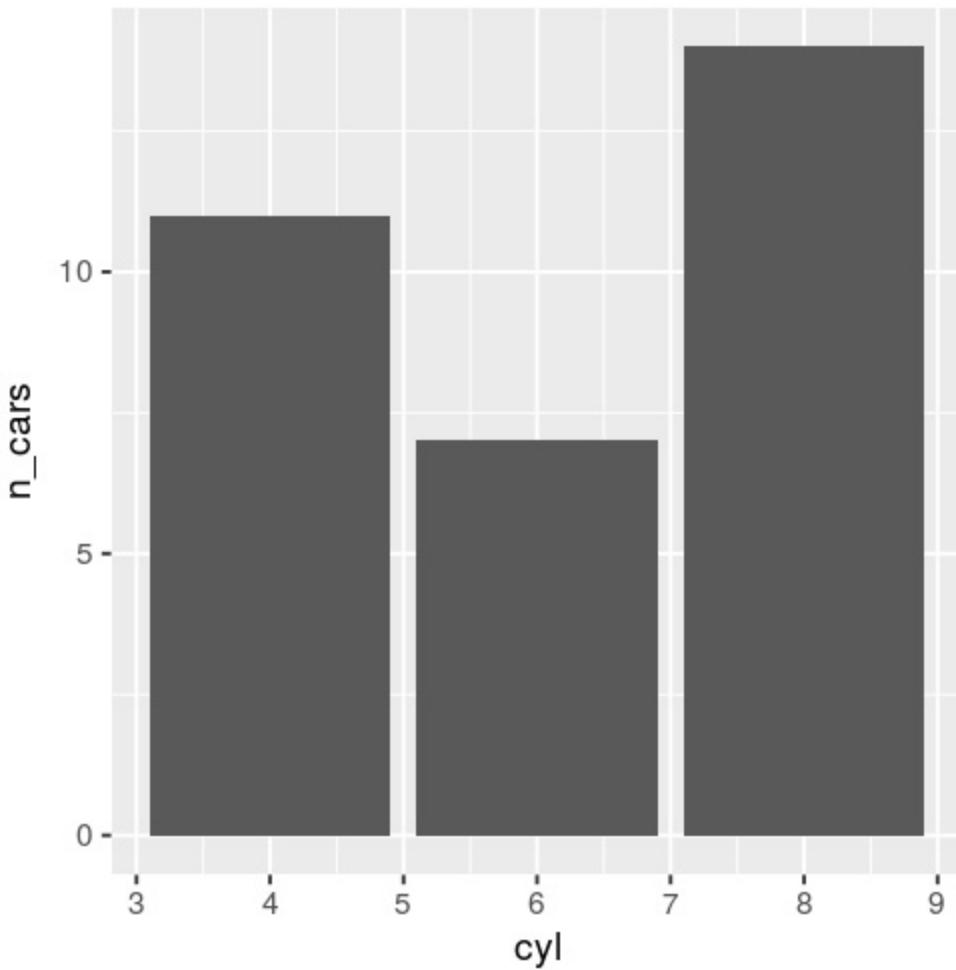
```
ggplot(average_mpg_by_cyl, aes(cyl, n_cars)) +  
  geom_bar(stat = "identity")
```

There is a useful shortcut for this. The `geom_col()` plots a column with a specified height (figure 9.27):

```
ggplot(average_mpg_by_cyl, aes(cyl, n_cars)) + geom_col()
```



**Figure 9.26** An identity statistic indicates that counts don't need to be calculated.



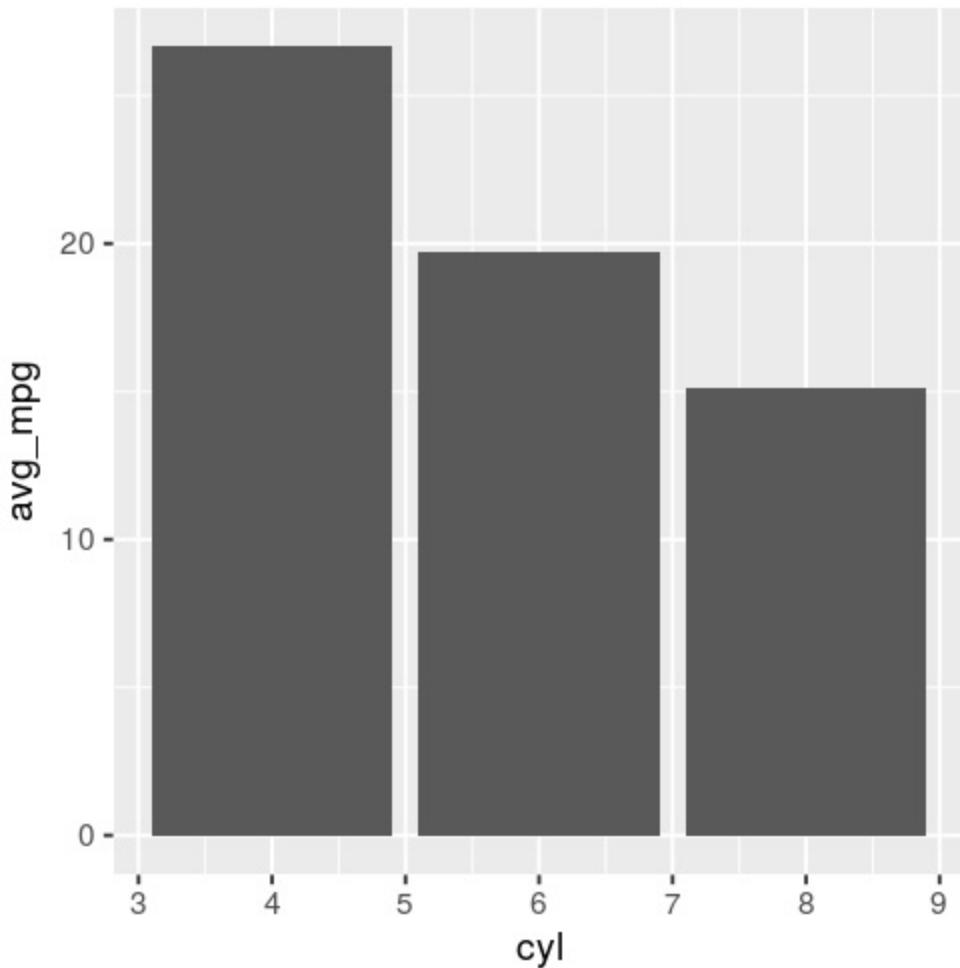
**Figure 9.27** `geom_col()` is useful if you know the heights of the bars.

In that case, you can plot any quantity for which you have calculated how tall you want the bar to be (for example, [figure 9.28](#)):

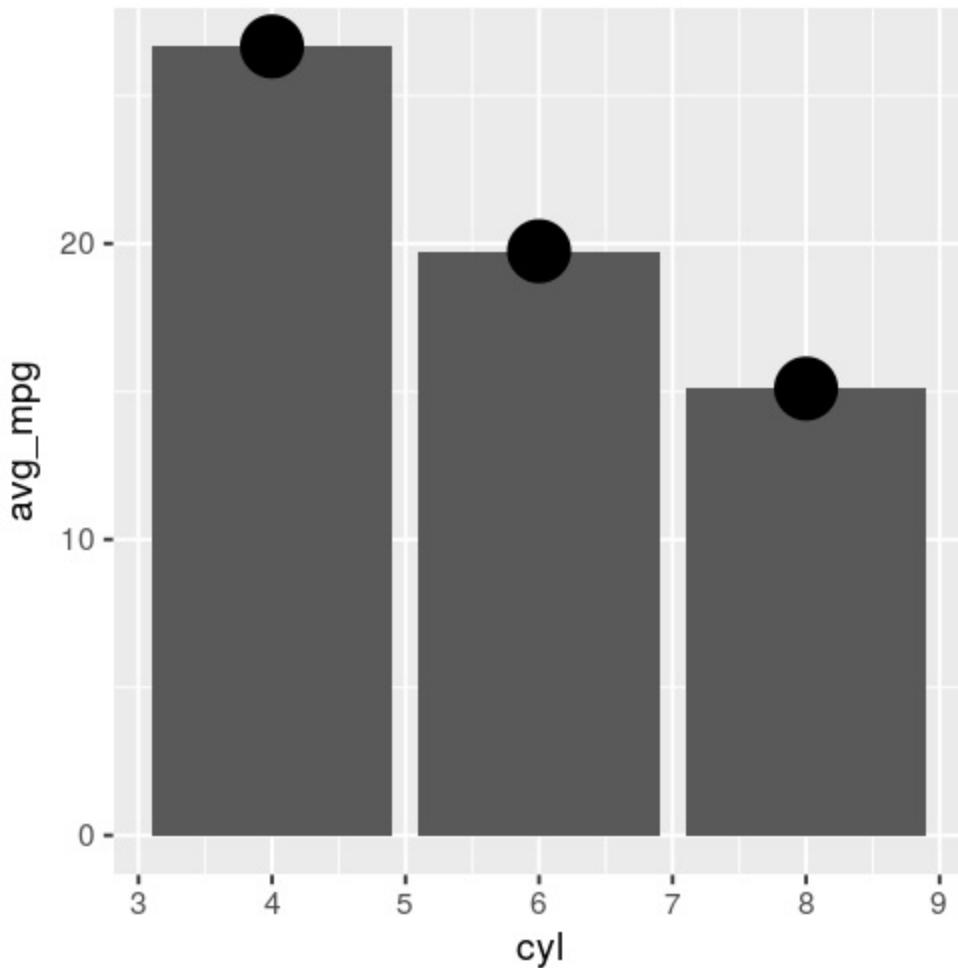
```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg)) + geom_col()
```

You can combine this with other `geom_s` too. If you want the top of the bar to have a point on it, you can add that, making use of the inherited `aes()` ([figure 9.29](#)):

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg)) +
  geom_col() +
  geom_point(size = 8)
```



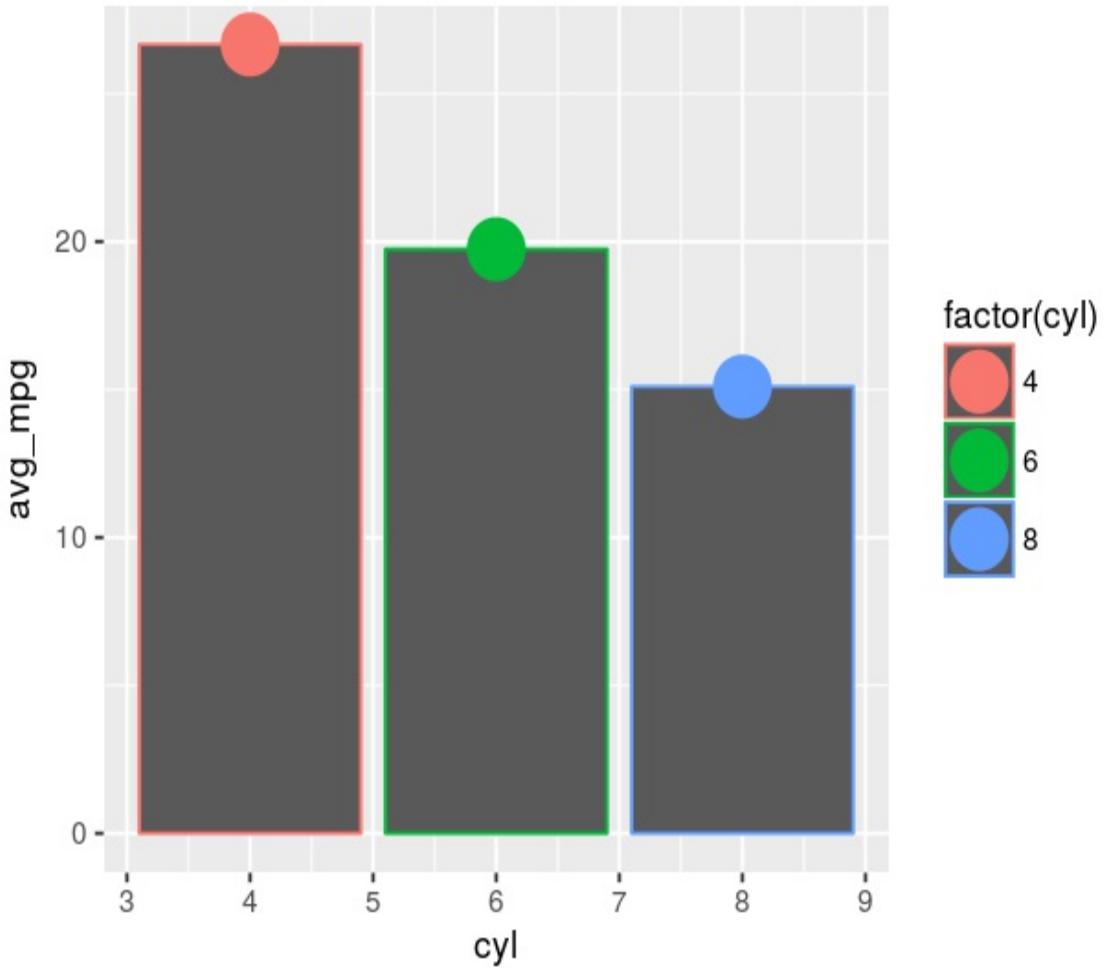
**Figure 9.28** Another `geom_col()` plot



**Figure 9.29 Adding points to a `geom_col()` plot**

Again, color can be added to more clearly delineate the categories. There are two options to do this. Trying to adapt the previous code using `col` changes the line color for the `geom_col()` (figure 9.30):

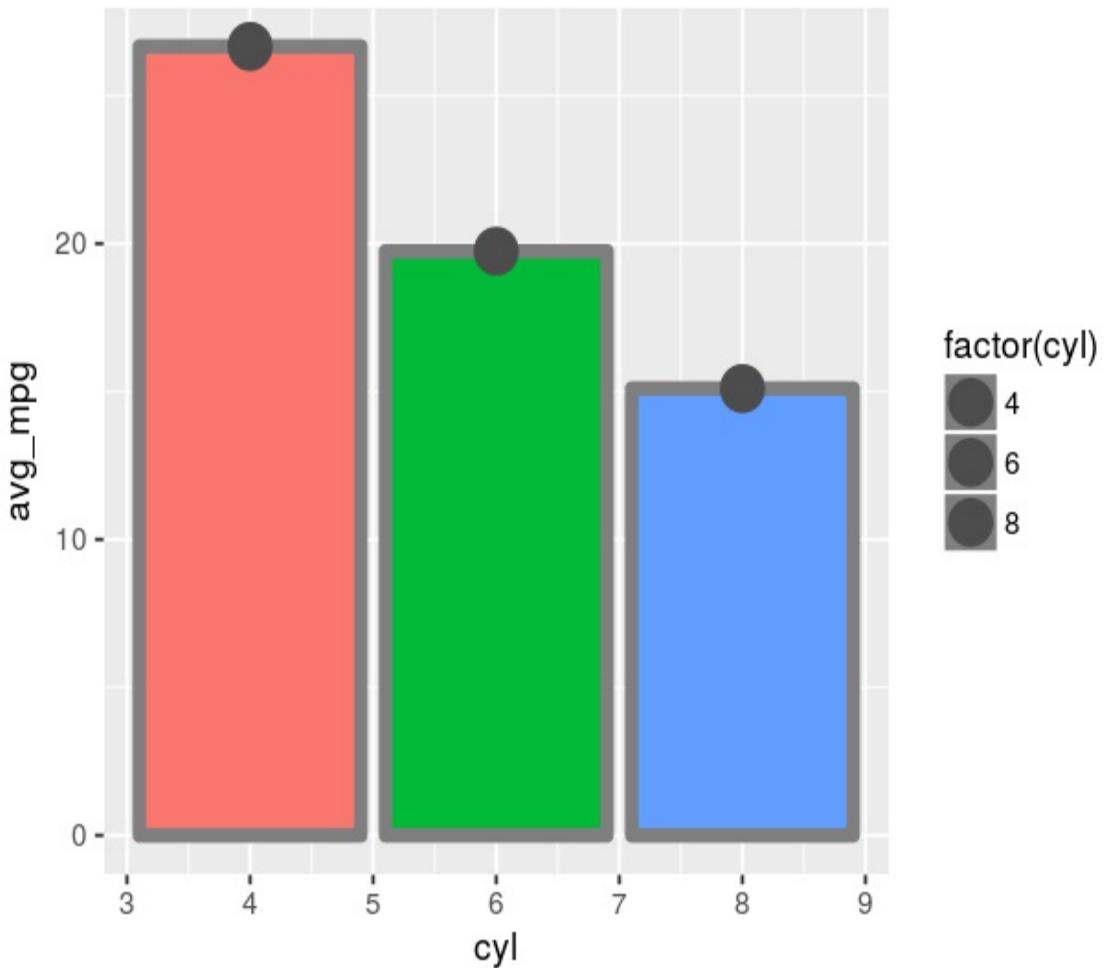
```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg, col =  
factor(cyl))) +  
  geom_col() +  
  geom_point(size = 8)
```



**Figure 9.30** The `col` aesthetic is inherited, and this changes the border line color for `geom_col()`.

To change the color of the fill, use the `fill` aesthetic. You can use this in tandem with the `col` aesthetic to produce some highlighting (for example, in figure 9.31):

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg, fill =  
factor(cyl))) +  
  geom_col(col = "grey50", lwd = 2) +  
  geom_point(col = "grey30", size = 6)
```



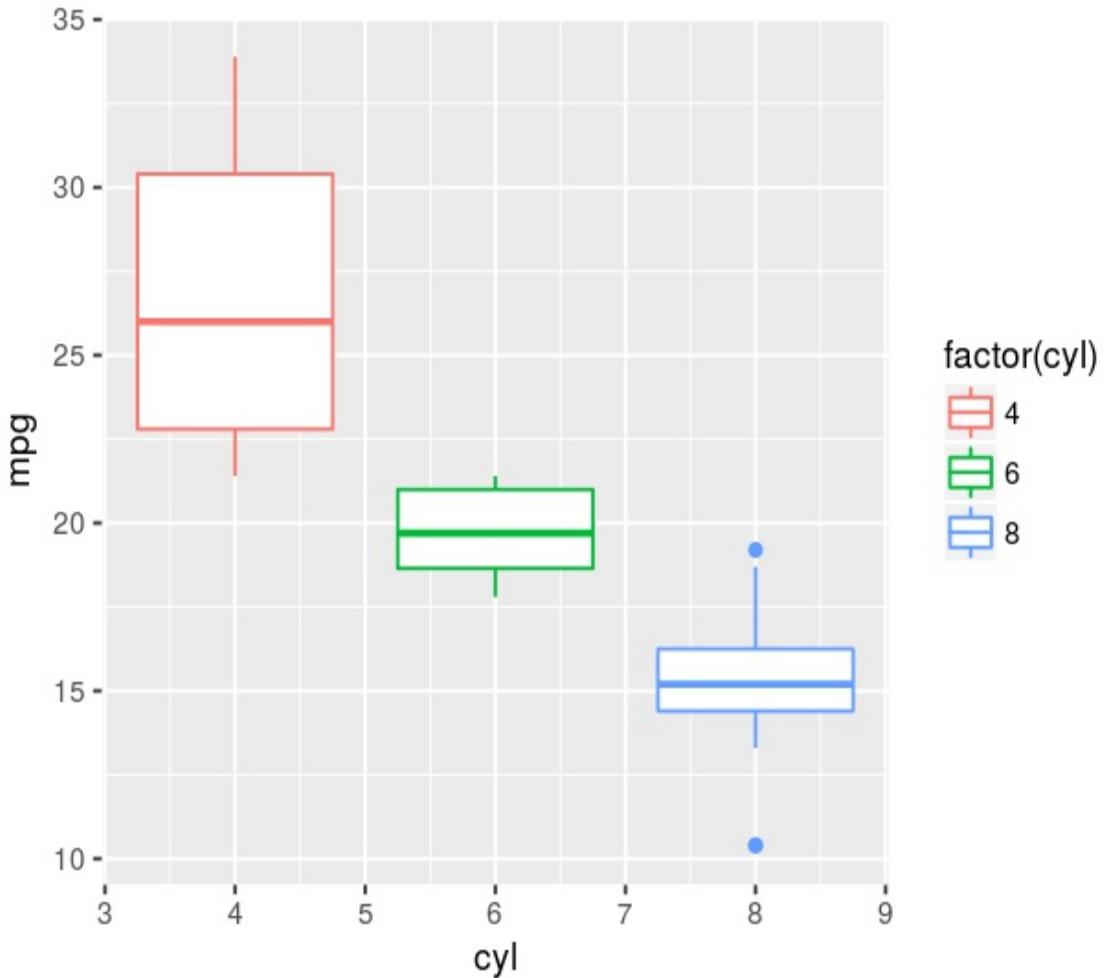
**Figure 9.31** The `fill` aesthetic changes the internal color of the bars.

### 9.2.6 OTHER TYPES OF PLOTS

To try to list all of the different types of plots made available by `ggplot2` and its extensions would require its own book (and several such books already exist). Instead, I'll point you in the direction of a few common and useful `geom_`s.

When plotting data that varies within a group, it can be useful to show the extent of that variability. A box-and-whiskers plot (a *boxplot*, shown in figure 9.32) can be useful in that case:

```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_boxplot()
```



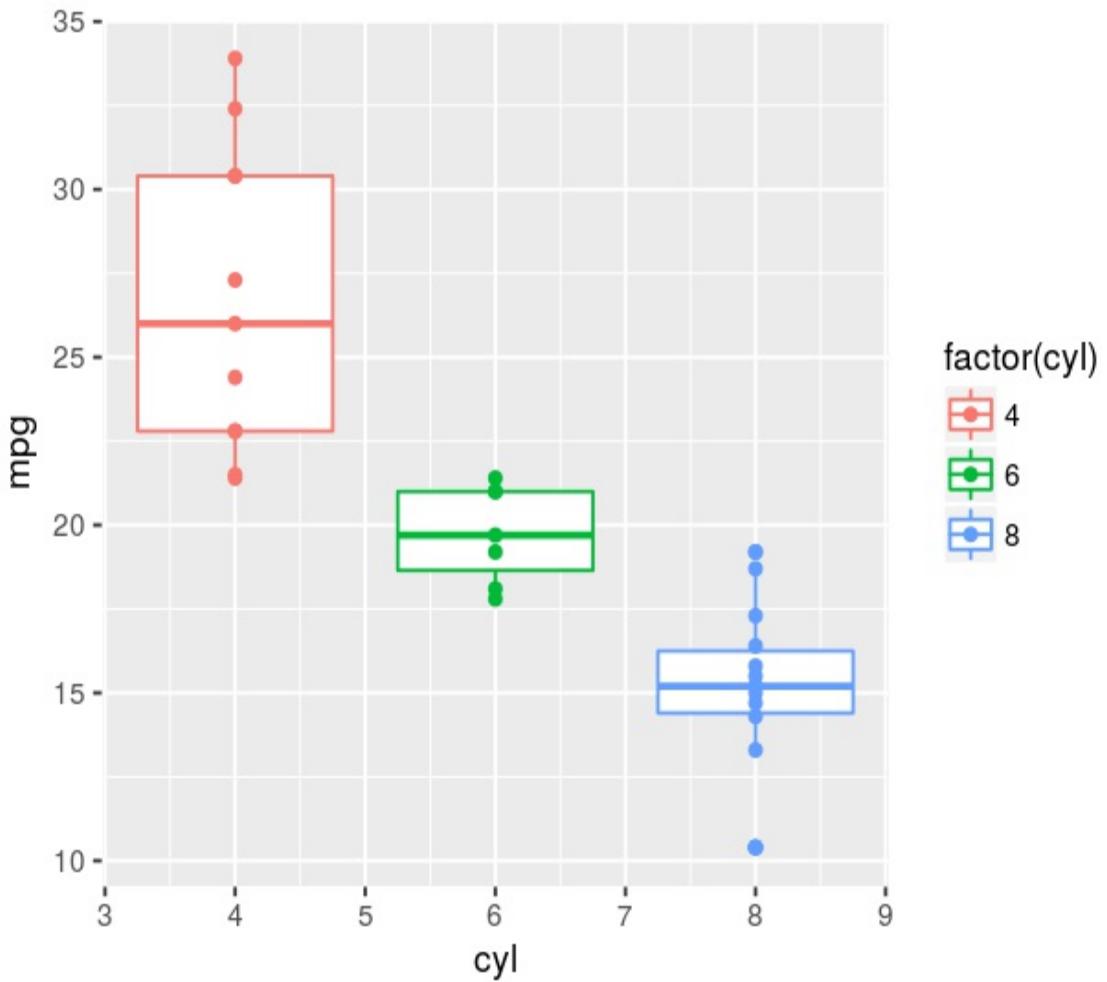
**Figure 9.32** A boxplot shows the data extent within a category.

The definitions for the edges of the boxes and the whiskers are found in `?geom_boxplot`, but the interpretation is fairly standard. What this doesn't show (well) is the distribution of points in the range. You can add the points (see [figure 9.33](#)):

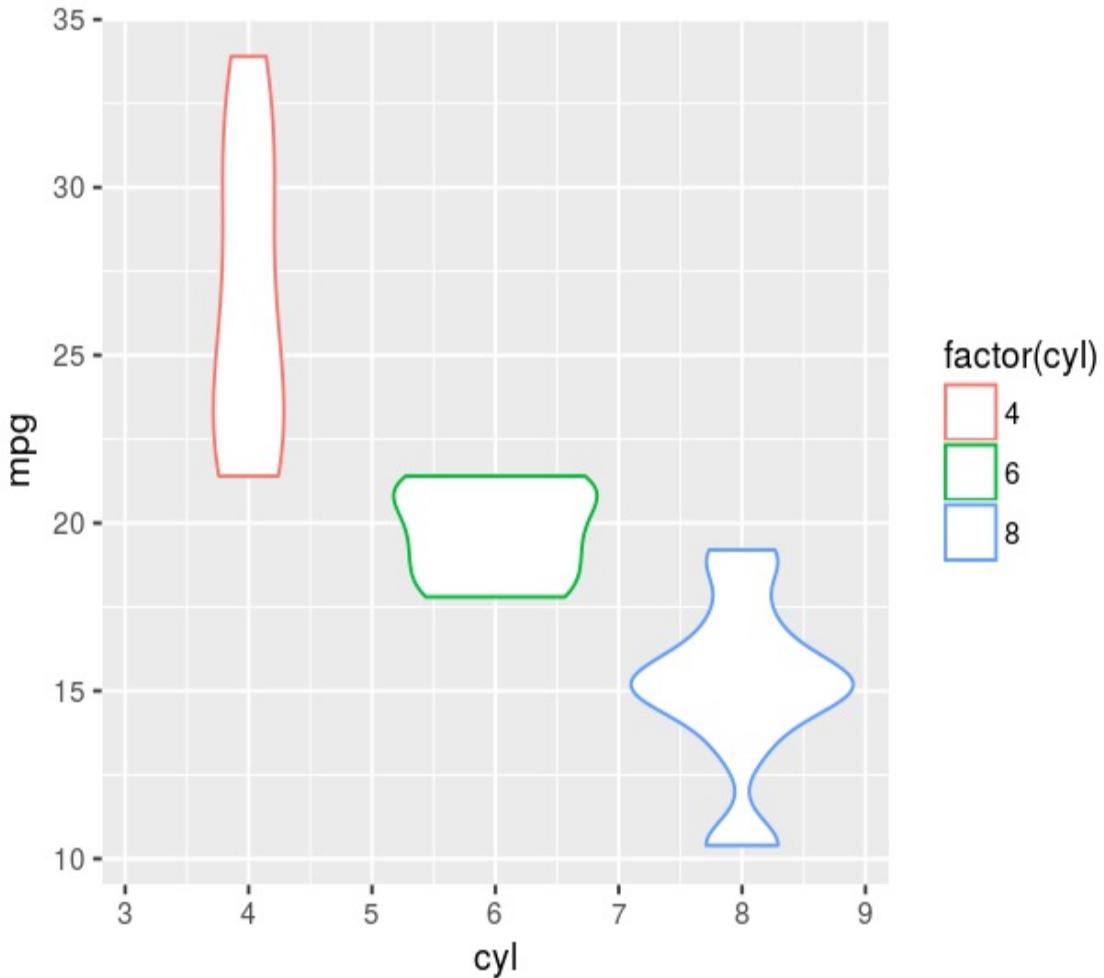
```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_boxplot() +
  geom_point()
```

This is perhaps better handled by a *violin* plot (see [figure 9.34](#)), which illustrates the density of points:

```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_violin()
```



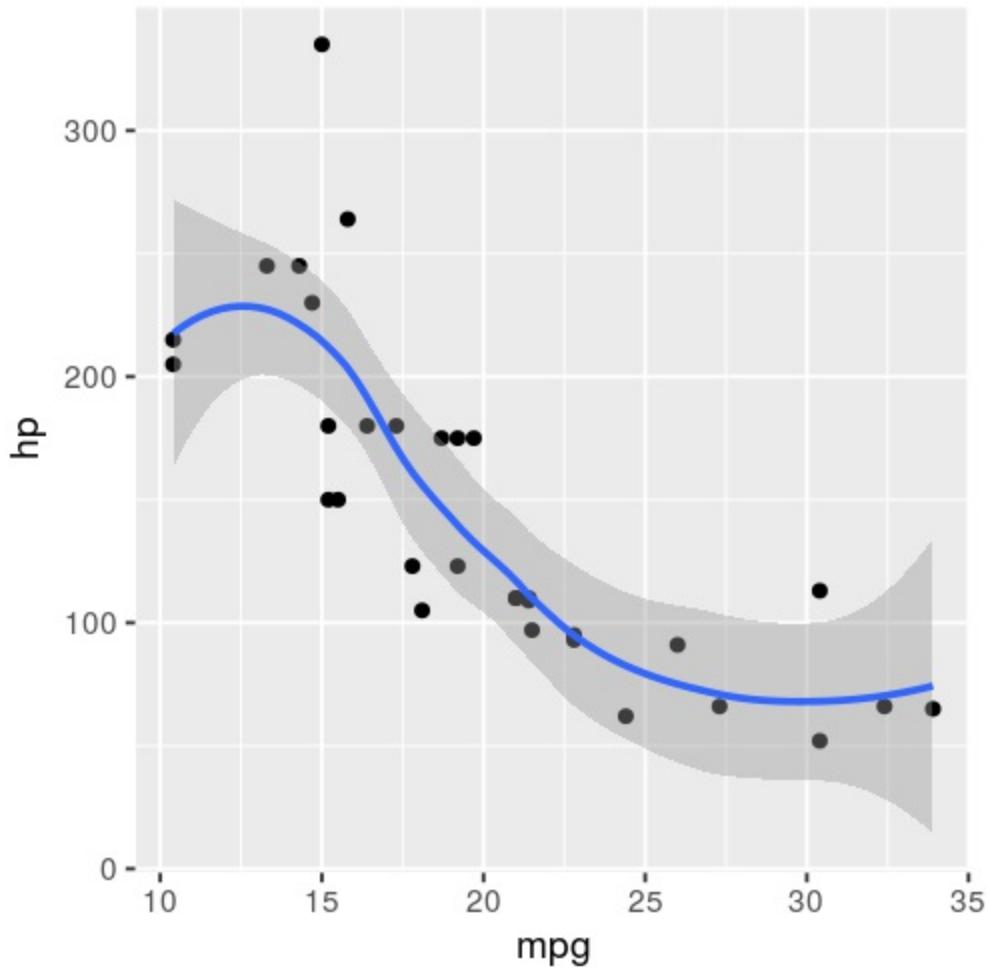
**Figure 9.33 A boxplot with points overplotted**



**Figure 9.34** A violin plot extends the notion of a boxplot by showing the distribution of plots within each category.

You can show the general trend of some points (with or without the points themselves) using a *smoothing* function that calculates the local trend at any point (see figure 9.35):

```
ggplot(mtcars, aes(mpg, hp)) +  
  geom_point() +  
  geom_smooth()  
# `geom_smooth()` using method = 'loess'
```



**Figure 9.35** A smoothing function shows the local trend at any point (with confidence interval shading).

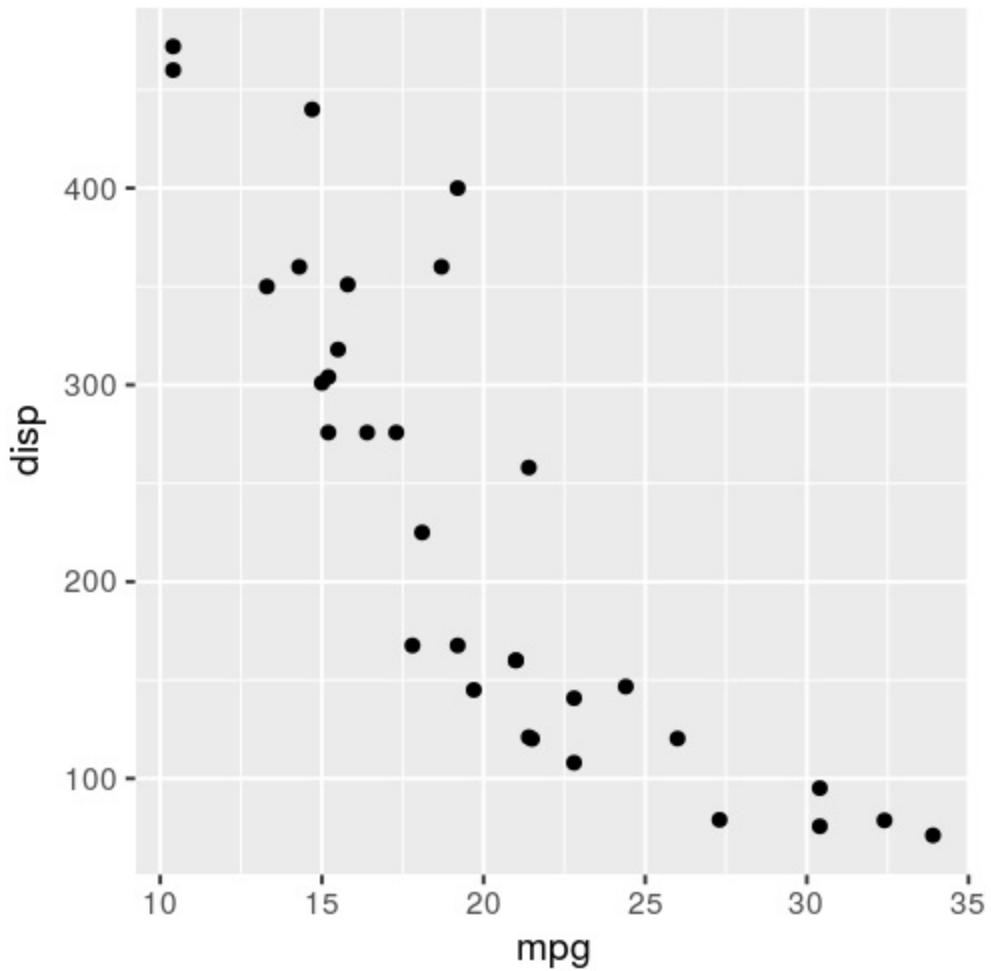
The message produced tells about the type of smoothing used and the assumed formula, and even though this is usually what you want, a different smoothing function can be specified explicitly. There are many ways to present data, and part of communicating it well is finding the right representation.

### 9.2.7 SCALES

So far, when you've asked `ggplot2` to handle data values as aesthetics (for example, `x = cyl` or `color = factor(am)`), you've trusted that it can decide how to do that. The range of the x-axis and the colors to be used are chosen by a default selection process. That won't always be what you want, though.

To change the range over which some feature is presented, you need the fourth ingredient in the “grammar of graphics”: *scale*, accessed using `scale_` functions. The simplest notion of changing a scale is the range of the x- or y-axis. You can do that using either `scale_x_continuous()` or `scale_x_discrete()`, depending on whether the data is of either of those definitions. You may have noticed that by default, `ggplot()` adjusts the axes ranges to roughly the extent of the data in each direction. Our go-to plot (see figure 9.36) doesn’t start at  $x = 0$ :

```
ggplot(mtcars, aes(mpg, disp)) + geom_point()
```

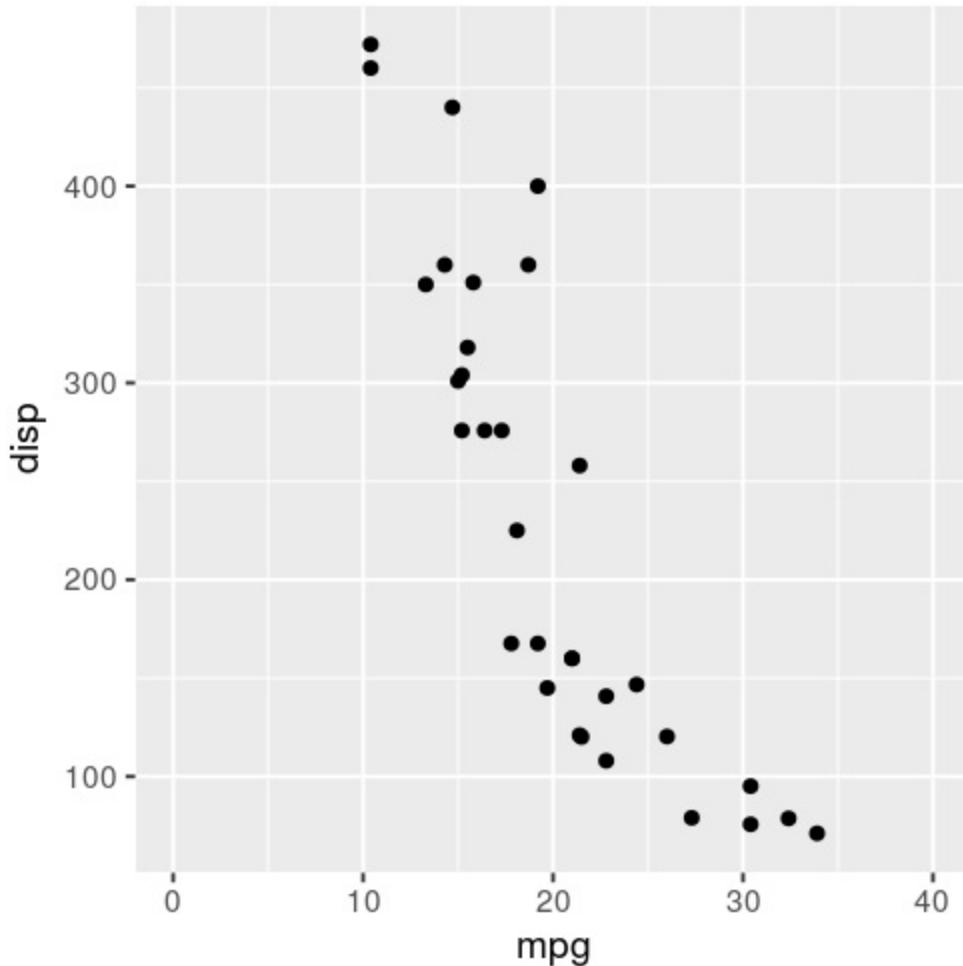


**Figure 9.36** Note that the axes don’t start at 0; they have been chosen to emphasize the existing data.

To enforce that (given that the x values, `mpg` are continuous), use

a vector to describe the start and end limits, as shown in figure 9.37:

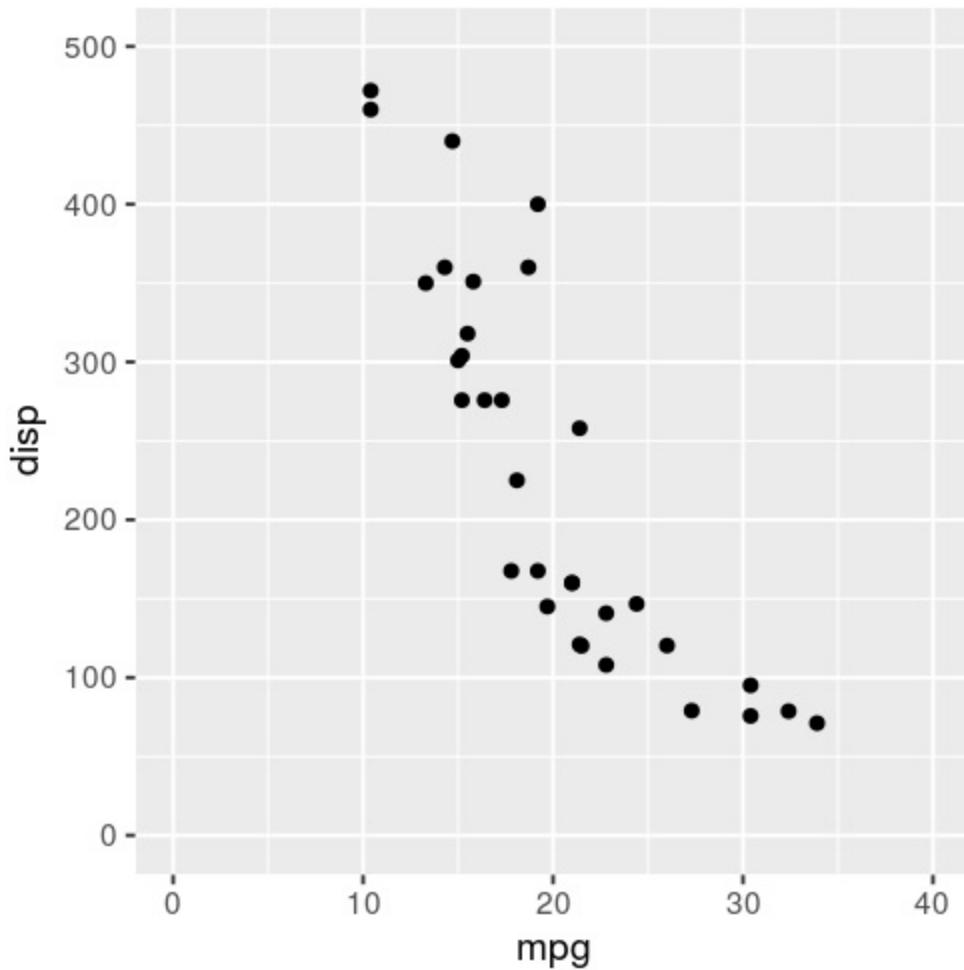
```
ggplot(mtcars, aes(mpg, disp)) +  
  geom_point() +  
  scale_x_continuous(limits = c(0, 40))
```



**Figure 9.37** Changing the x-axis scaling with `scale_x_continuous` to start the range at 0

You can do the same for the y-axis (`disp`, also continuous), as shown in figure 9.38:

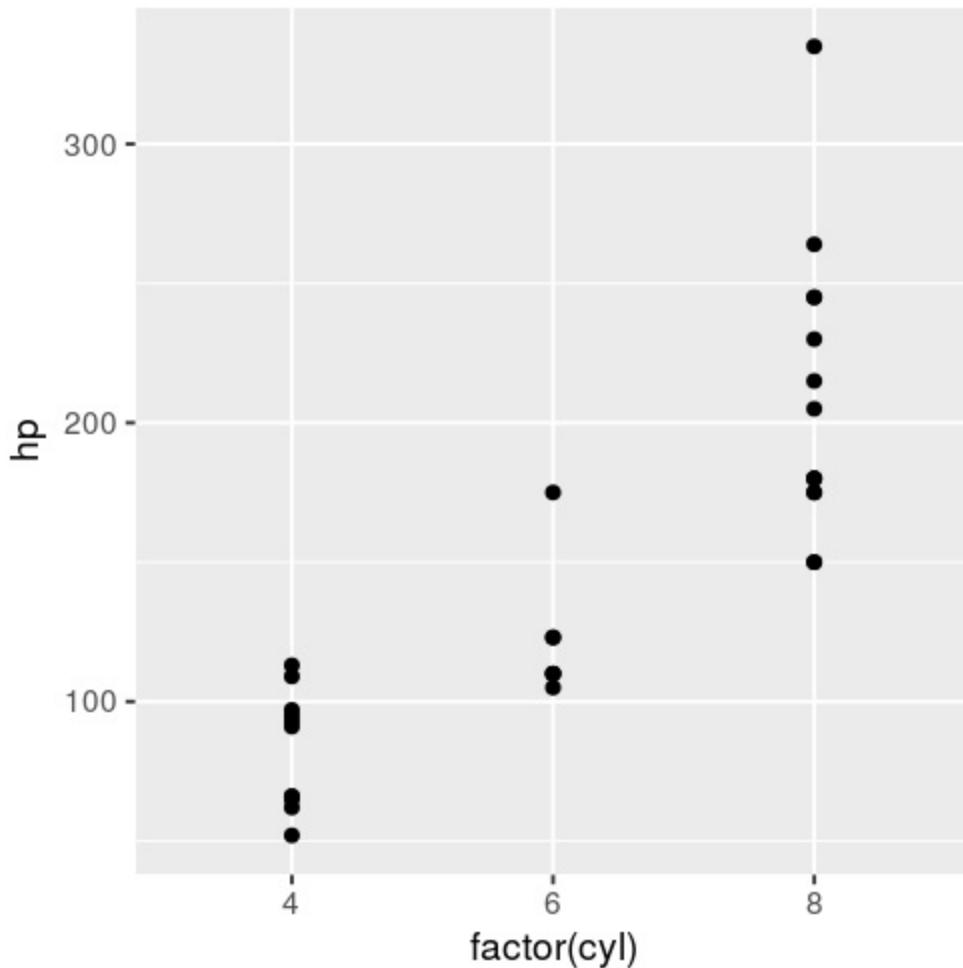
```
ggplot(mtcars, aes(mpg, disp)) +  
  geom_point() +  
  scale_x_continuous(limits = c(0, 40)) +  
  scale_y_continuous(limits = c(0, 500))
```



**Figure 9.38** Both axes can be scaled however you decide.

When the data is discrete, you use an alternative `scale_` function. For example, if you treat `cyl` as a factor, you may want to change the scale to include potential two-cylinder vehicles (additional data on a Fiat 500, perhaps). Originally (see figure 9.39), you have just the three factor levels present in the data:

```
ggplot(mtcars, aes(factor(cyl), hp)) +  
  geom_point()
```



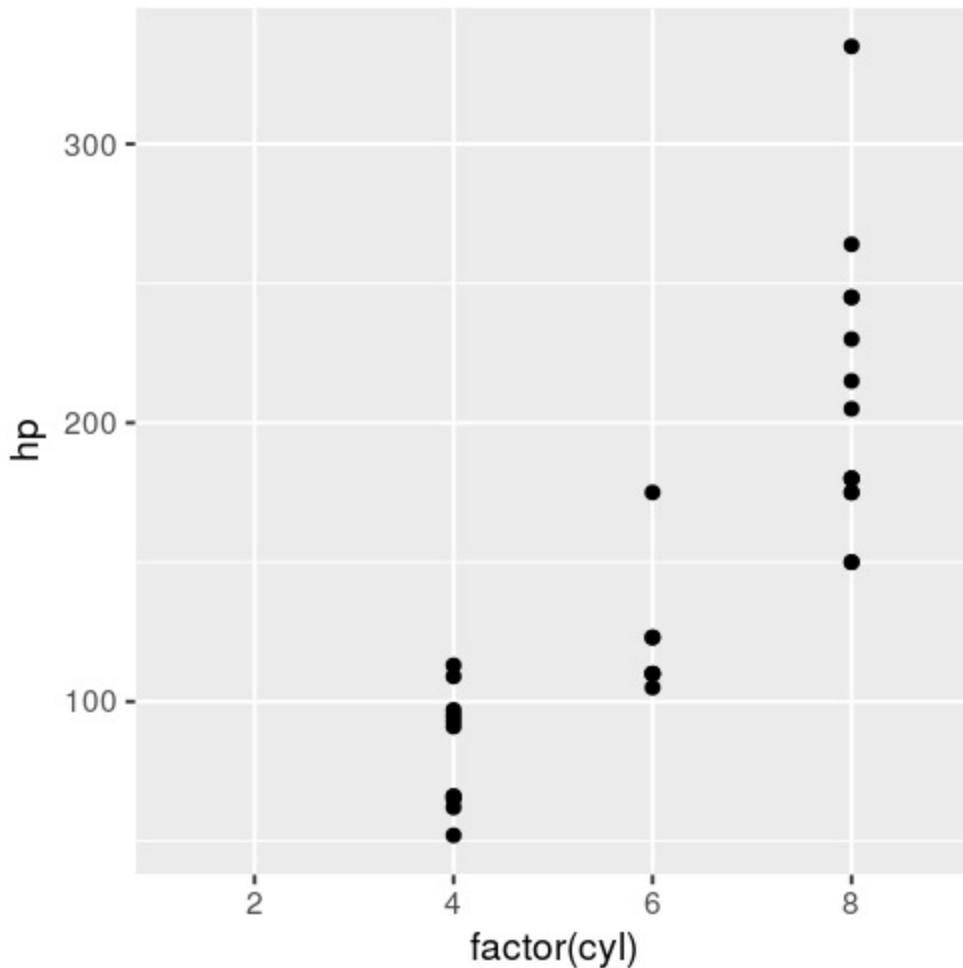
**Figure 9.39** This shows the three existing levels in cyl.

But you can extend that, as has been done in [figure 9.40](#):

```
ggplot(mtcars, aes(factor(cyl), hp)) +  
  geom_point() +  
  scale_x_discrete(limits = as.character(seq(2, 8, by =  
  2))) ①
```

①

In this example, the categories are factor levels, so the limits also need to be strings.



**Figure 9.40** You can extend the scale to a not-yet-present fourth level.

This usage of `limits` differs somewhat from the usage in `scale_x_continuous()`. The help file for `scale_x_discrete()` notes that the meaning of the argument `limits` is “A character vector that defines possible values of the scale and their order.” That’s what you’ve provided — a character vector (due to `as.character()`) of the values to be used on the x-axis, and their order. These are `c("2", "4", "6", "8")`.

## DISCRETE SCALES

The spacing of factors on an axis is, by default, even. The example of counting `cyl` by twos means that it looks clean on the axis, but you may have an uneven spacing of factor levels, say, if the first level was “1” and not “2”, as in figure 9.41:

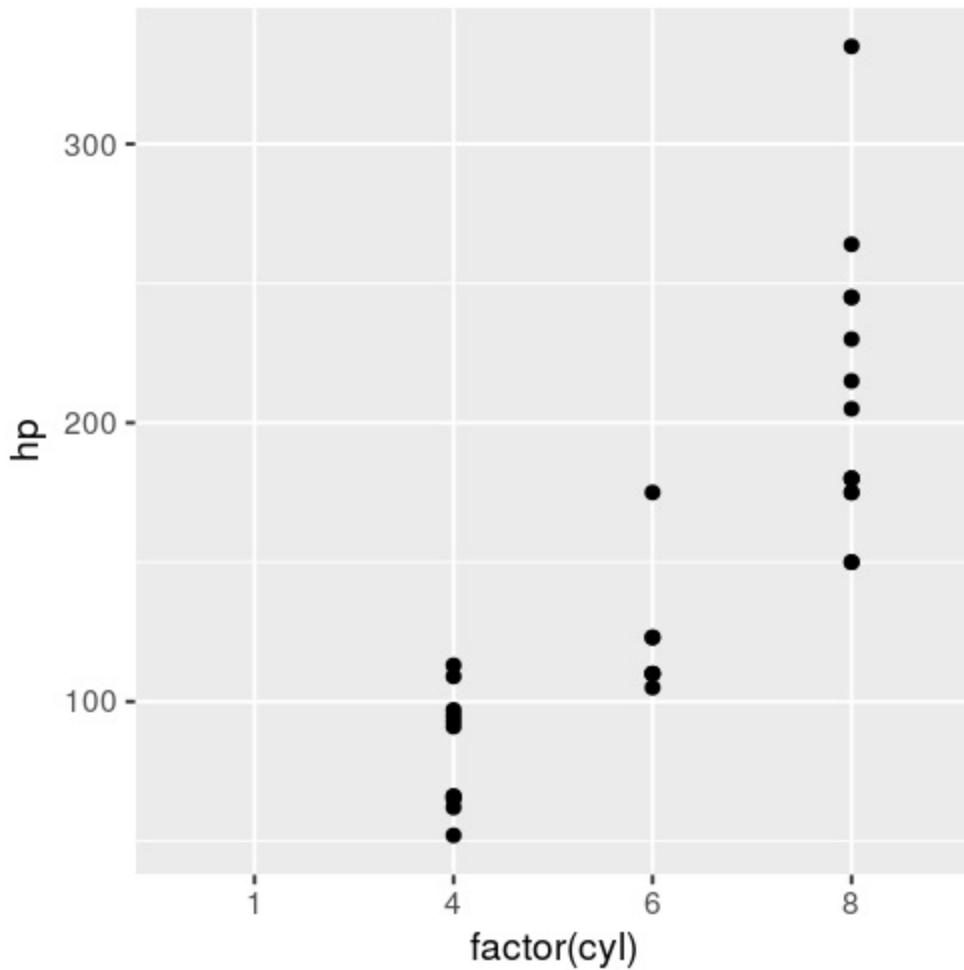
```
ggplot(mtcars, aes(factor(cyl), hp)) +  
  geom_point() +  
  scale_x_discrete(limits = as.character(c(1, 4, 6, 8)))
```

That looks less clean. Remember, though, you're defining these as categories, not as numbers, so the spacing between them should be equal. Similarly, if the categories didn't resemble numbers, it would make just as much sense to have them equally spaced. Here is some data with three factor levels:

```
gifts <- data.frame(  
  gift = c("French\nHens", "Turtle\nDoves",  
  "Calling\nBirds"), ①  
  n_items = c(3, 2, 4)  
)
```

①

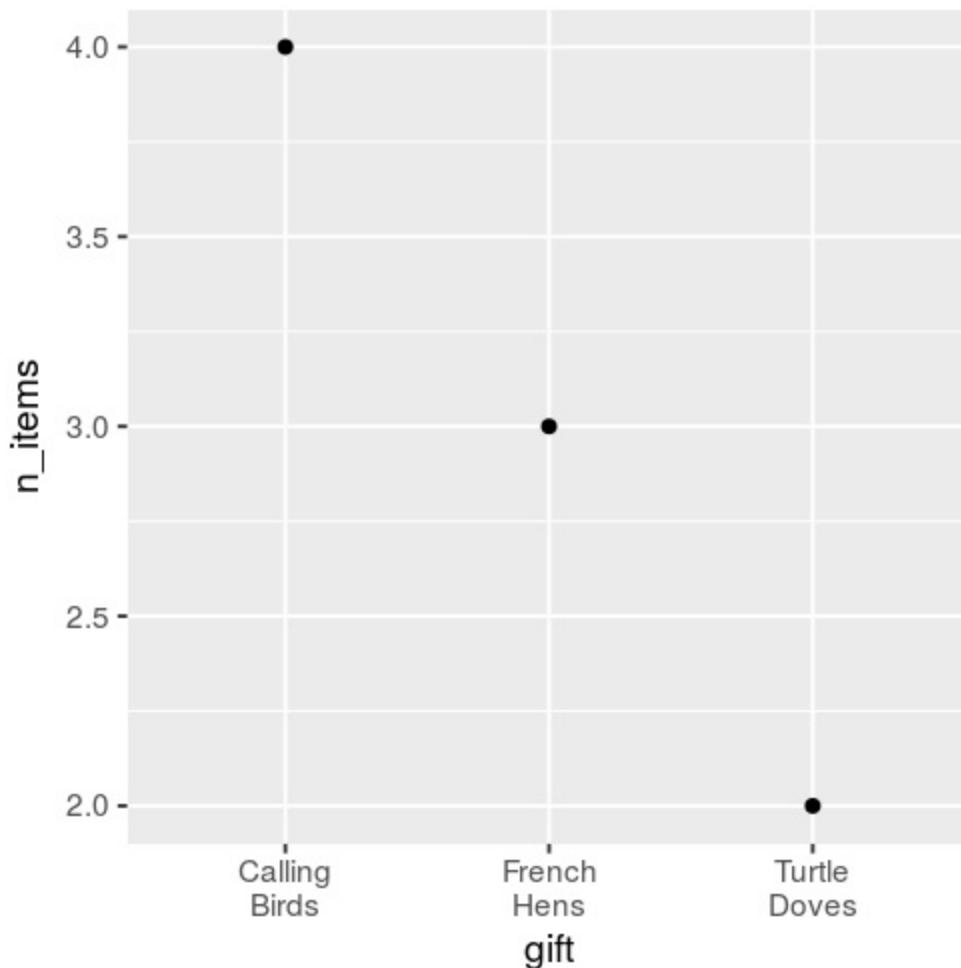
**Remember, factor levels can contain escaped characters — see what ggplot() does with them.**



**Figure 9.41** Categories appear at even spacing along the axis by default.

You can plot them with the default range (see figure 9.42):

```
ggplot(gifts, aes(gift, n_items)) +  
  geom_point()
```



**Figure 9.42 Three categories (factor levels) plotted**

This is just the data for that small subset of what will eventually be all of the data. Rather than creating empty data in a clean dataset, you can specify that the limits need to be expanded by providing the full list of options, resulting in figure 9.43:

```
all_gifts <- c(
  "Partridge\nin\na\nPear\nTree",
  "Turtle\nDoves",
  "French\nHens",
  "Calling\nBirds",
  "Gold\nRings",
  "Geese\nna-Laying",
  "Swans\nna-Swimming",
  "Maids\nna-Milking",
  "Ladies\nDancing",
  "Lords\nna-Leaping",
```

```
"Pipers\nPiping",
"Drummers\nDrumming"
)

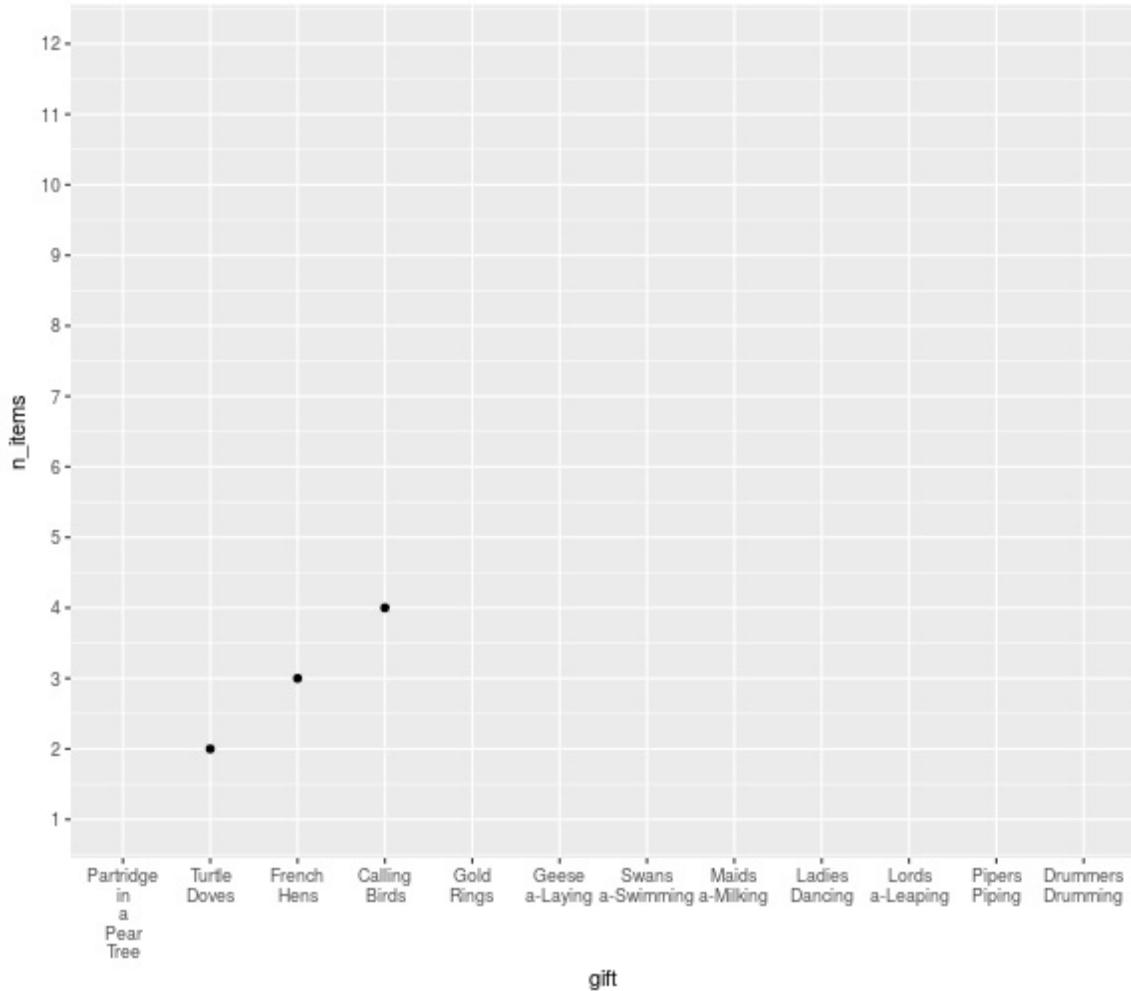
ggplot(gifts, aes(gift, n_items)) +
  geom_point() +
  scale_x_discrete(limits = all_gifts) + ①
  scale_y_continuous(limits = c(1, 12), breaks =
1:12) ②
```

①

The levels and their ordering are provided using the all\_gifts vector.

②

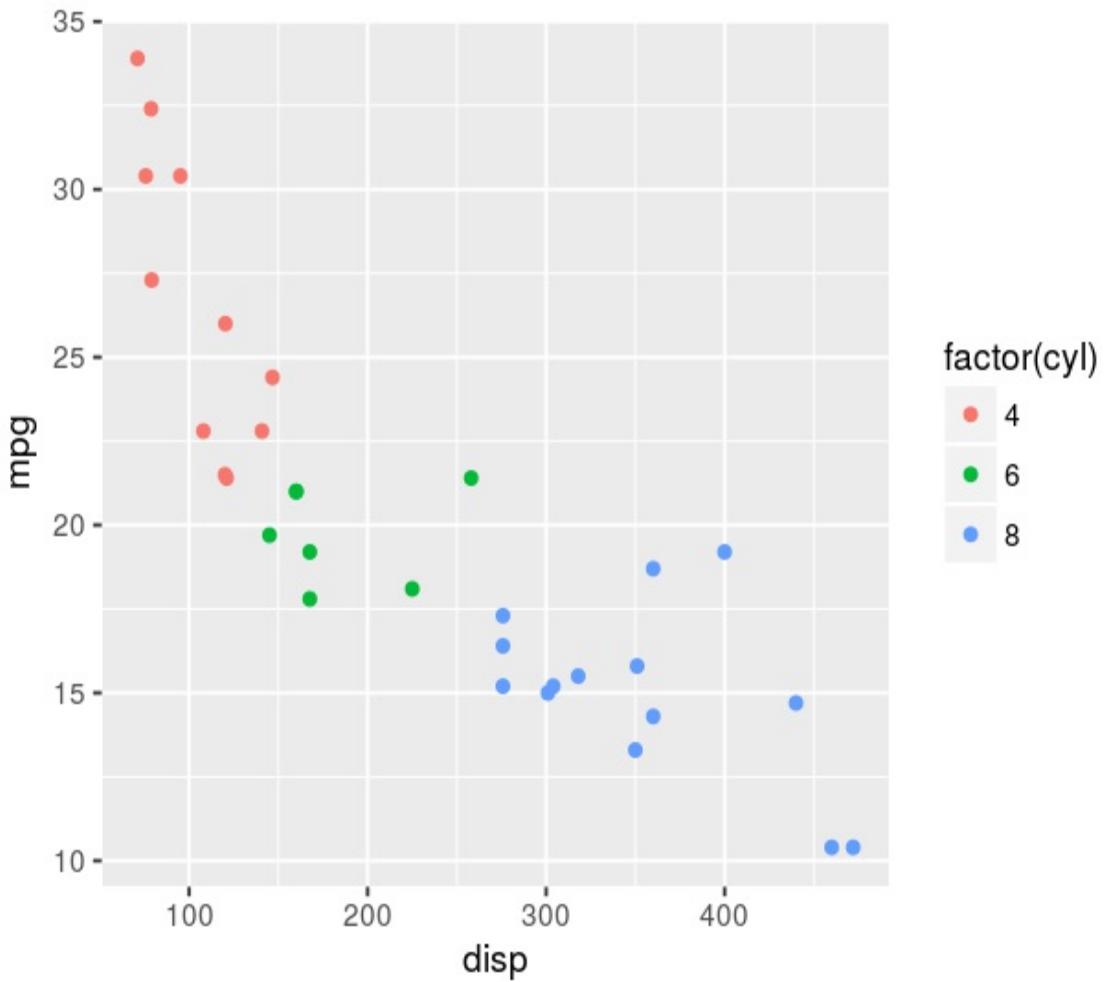
The argument breaks specifies not to use fractional tick marks on the y-axis.



**Figure 9.43** You can provide more levels in the scale even if they don't yet exist in the data.

The axes aren't the only scale you can work with — any of the aesthetics potentially has a scale you can manipulate. The default set of colors that `ggplot()` uses aren't the only options. Return to the plot (figure 9.44) for which `col` was specified by the level of `cyl`:

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point()
```



**Figure 9.44 Default colors for three levels**

If you want those colors to be different, you can use the `scale_color_discrete()` (or `scale_color_continuous()`) function. You can ask for the three colors to be chosen from the color wheel, but starting at a different point, by specifying the starting angle — say, 30 degrees, resulting in figure 9.45:

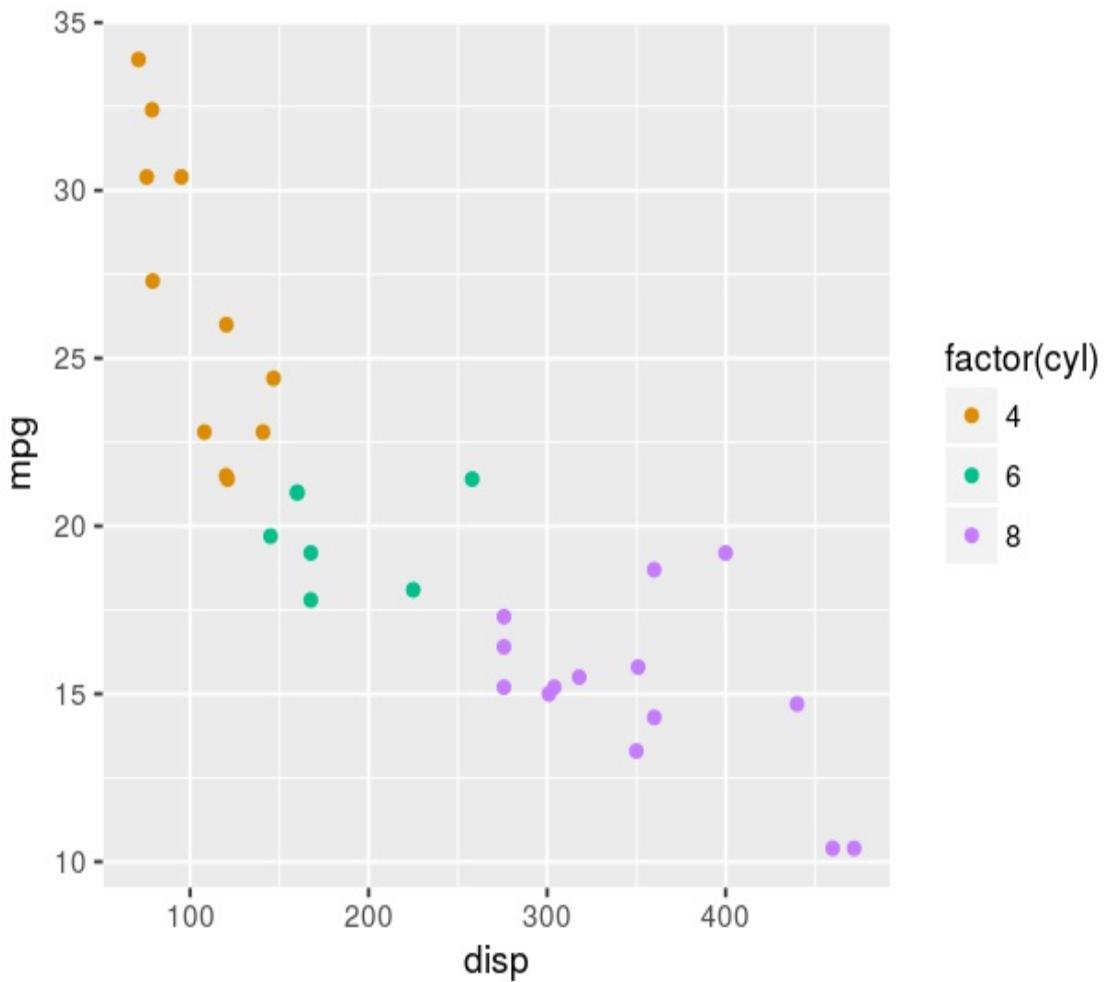
```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point() +
  scale_color_discrete(h.start = 30) ①
```

①

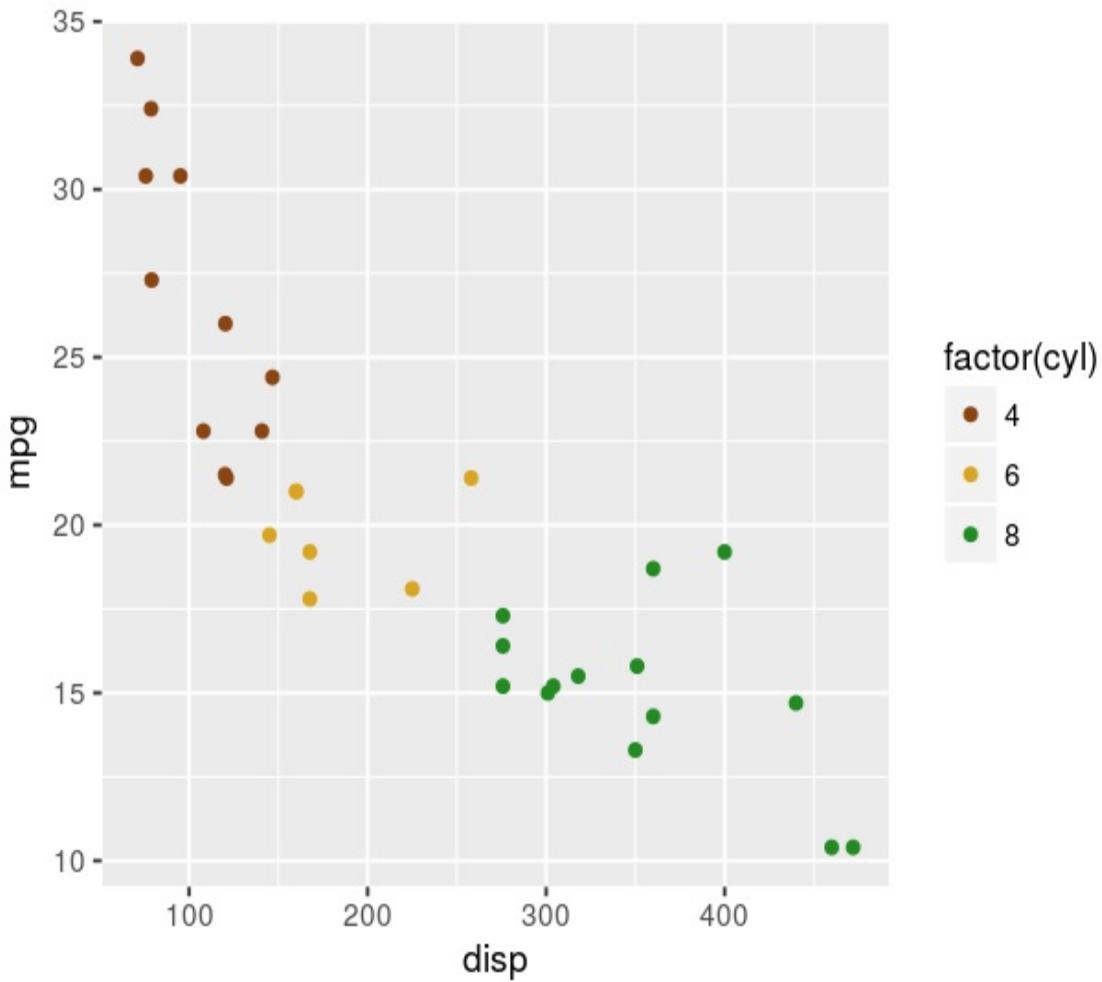
The default starting angle is 15 degrees.

Or you can specify the colors manually, as in figure 9.46:

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +  
  geom_point() +  
  scale_color_manual(values = c("saddlebrown", "goldenrod",  
"forestgreen"))
```



**Figure 9.45** Starting the color wheel at 30 degrees results in different colors.



**Figure 9.46** You can manually choose colors, too.

Many palettes of colors are available. Read `?scale_color_manual` for examples. There are similar `scale_` functions for most of the common aesthetics, and you should become familiar with as many of them as you can.

### 9.2.8 FACETTING

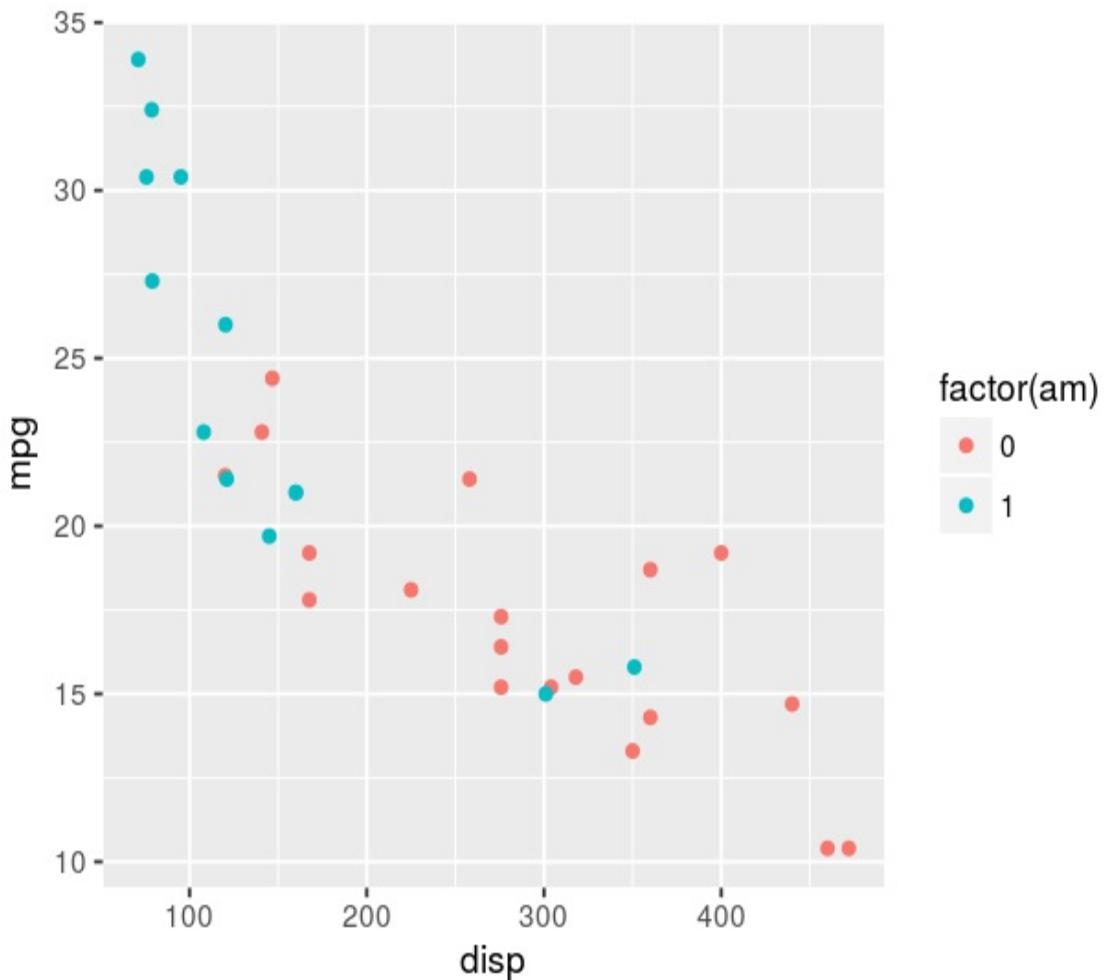
You've seen that aesthetics can distinguish characteristics, but this isn't always clear enough, especially if points or lines overlap considerably between groups. There's one more powerful feature of `ggplot2` that you can use to physically separate features based on some quantity: it's called *facetting*.

Rather than change an aesthetic of each group, you can create

individual plots for each group where only those values will be plotted. This is all handled automatically, so it's as simple as adding a call to `facet_grid()` or `facet_wrap()`, depending on how you want to lay out the individual plots.

Looking at the values of `mpg` as they vary by `disp`, you can color them according to the value of `carb`, resulting in figure 9.47:

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point()
```

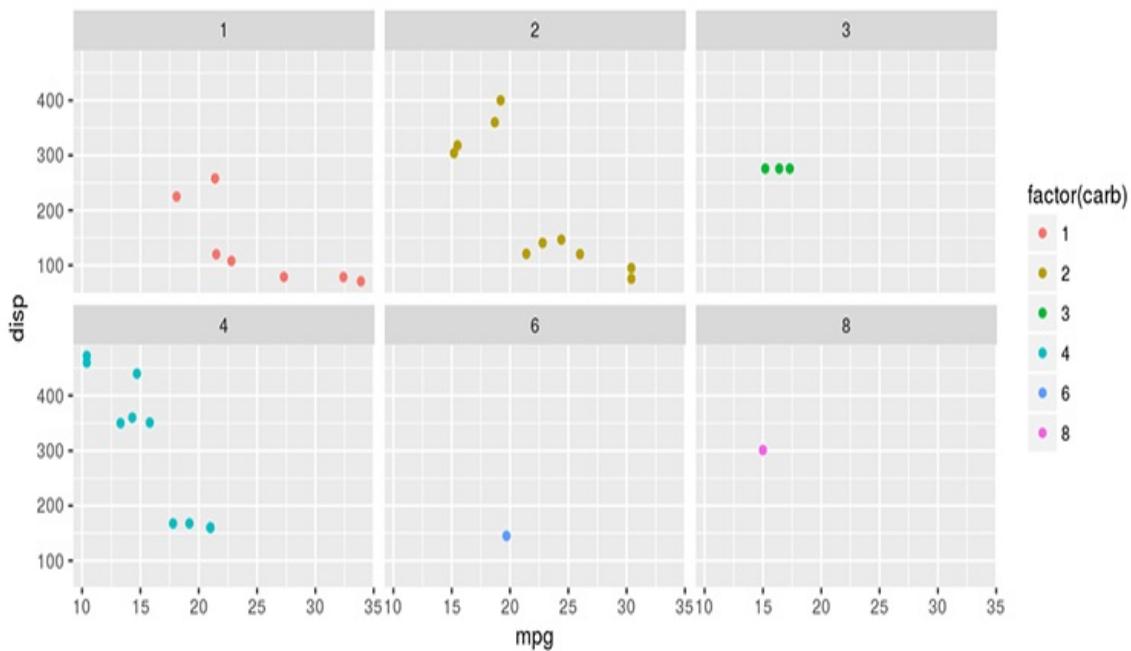


**Figure 9.47** One way to show another dimension is to use color to distinguish levels of a group.

But this category grouping isn't very clear; it's difficult to see what the relationship is between `mpg` and `disp` for each group. Instead, you can physically separate the different levels of `carb`

onto their own plots, as in [figure 9.48](#):

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_wrap(~ carb)
```



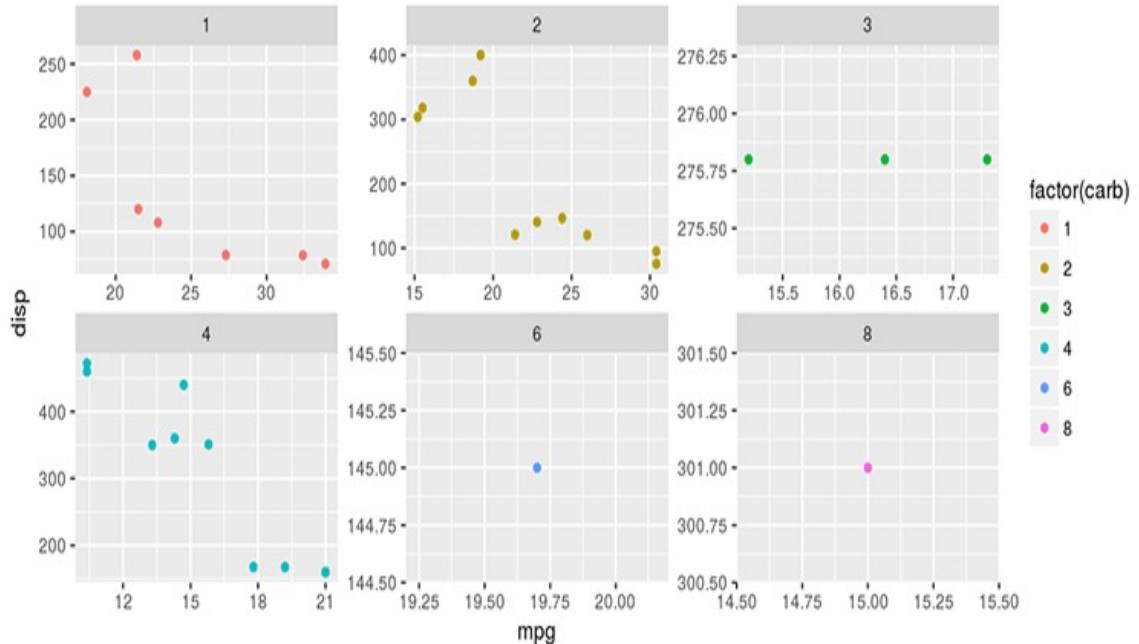
**Figure 9.48** Facetting moves each category level to its own plot.

The syntax for faceting is that of a *formula*. For `facet_wrap()`, a single-sided formula produces the effect you want. Equivalently, you can provide the argument as a character vector, so `facet_wrap("carb")` would produce the same result.

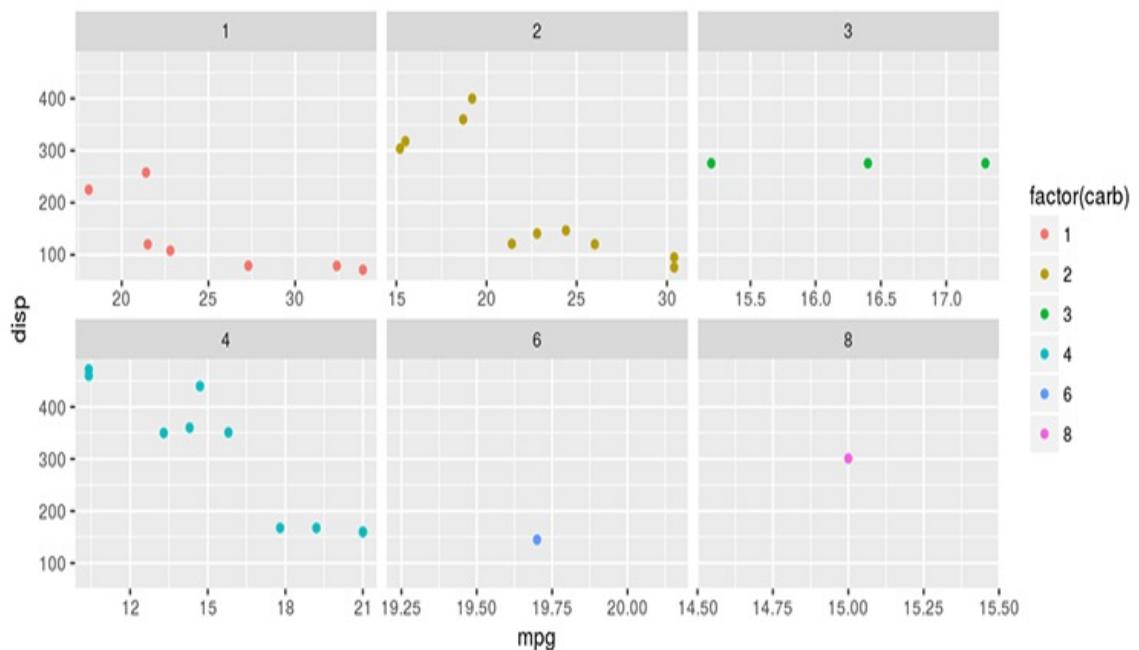
Note that each facet is labelled in the strip by the value of `carb` (values 1 through 8, missing 5 and 7) for that group. Individual relationships within that group are now clearer.

Some of the groups don't have many values, so it's sometimes useful to be able to change the scale. You need to change it differently for each facet, though, so that option is built right in as the `scales` argument, which takes a "free" value to make the scaling independent for each facet (shown in [figure 9.49](#)):

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_wrap(~ carb, scales = "free")
```



**Figure 9.49** Using `scales = "free"` means the scale in each facet panel is independent.



**Figure 9.50** You can keep only one axis fixed using `scales = "free_x"`.

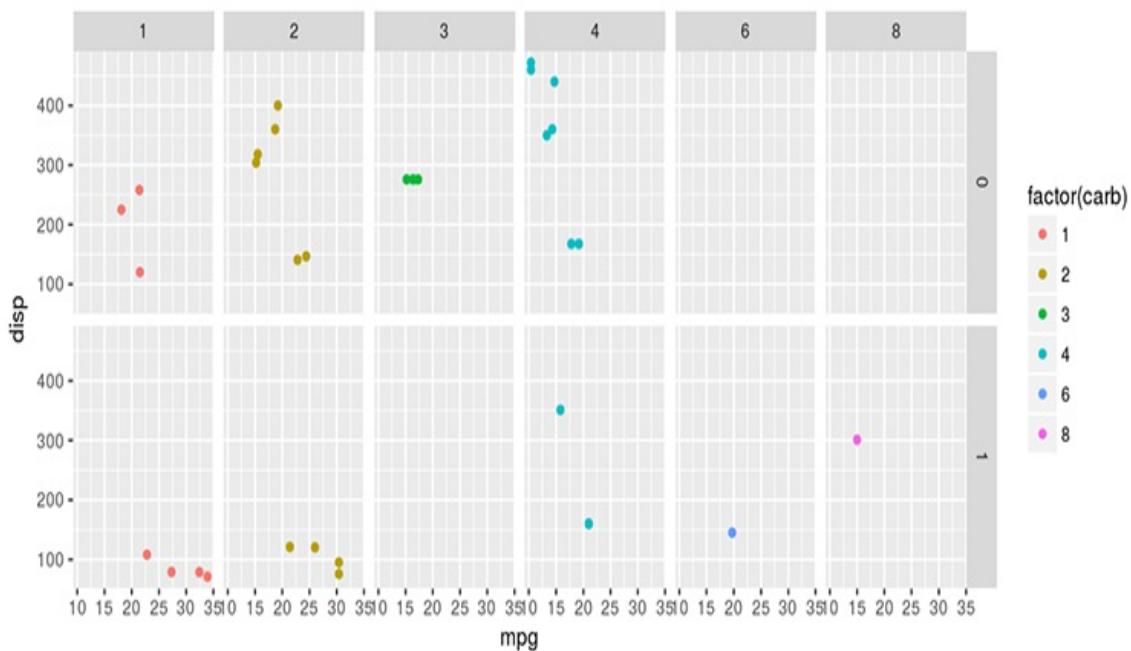
The other facetting function is `facet_grid()`, which combines two different features in a grid of facets. For example, if you think that the relationship between `mpg` and `disp` may behave

differently depending on both carb and am, you can investigate that and produce figure 9.51:

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_grid(am ~ carb)
```

Here the value of carb varies across a row of facets, while am varies between the columns of facets. Again, to allow the scales to vary, you can add that in and produce figure 9.52:

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_grid(am ~ carb, scales = "free")
```

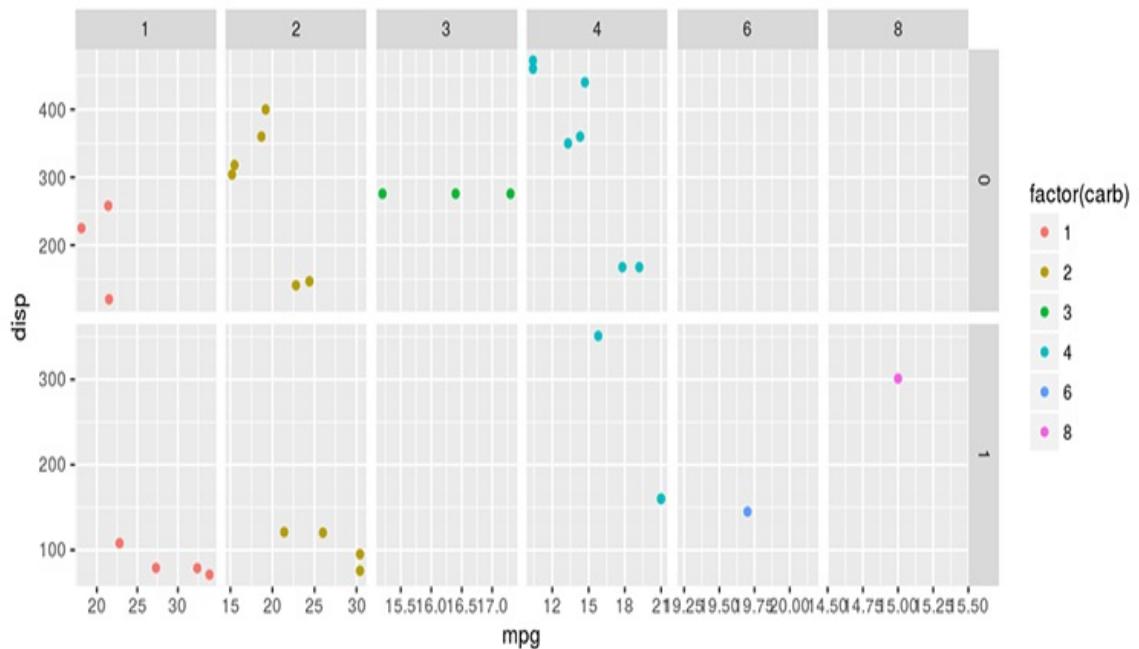


**Figure 9.51** You can make comparisons between two levels in a grid of values using `facet_grid()`.

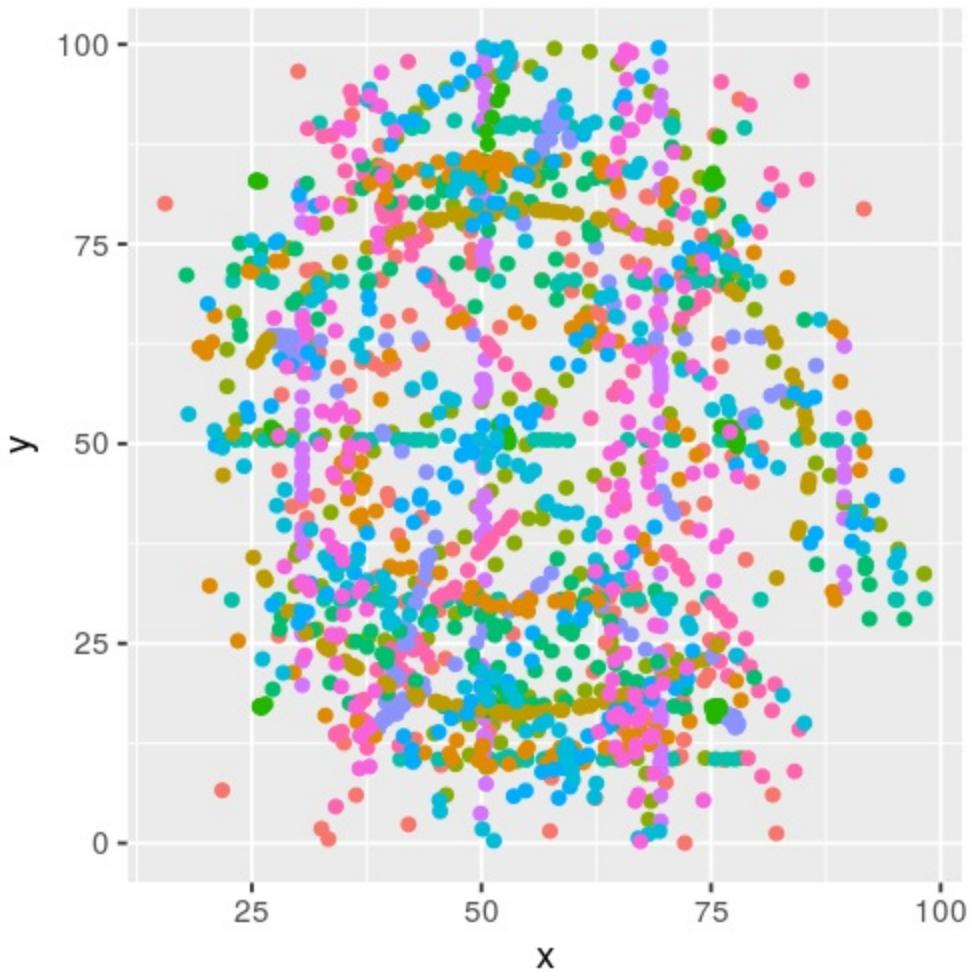
Facets at the intersections of valid values of carb and am with no data (for example, there is no data for `carb == 3` and `am == 1`) will still be shown when presented this way (the layout will always be a grid).

This is an extremely powerful tool that can give you insight into hidden patterns not otherwise visible. When you look at the overlap of a dozen datasets (colored by the dataset to which they

belong), it's impossible to tease out any patterns (see figure 9.53).



**Figure 9.52** Using the scales argument to make the axes independent

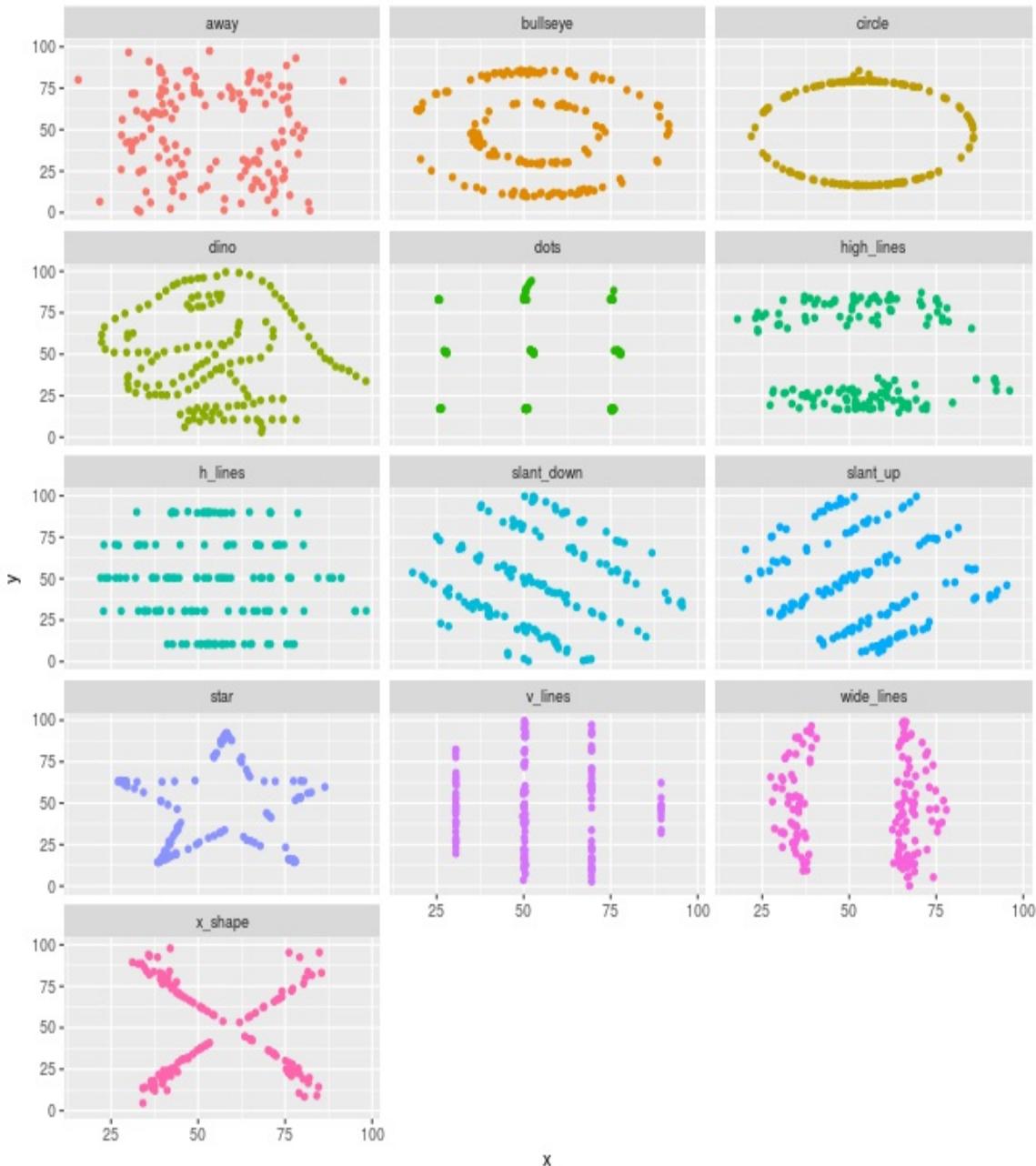


**Figure 9.53** Can you see the patterns in this plot? There are 13 different datasets.

Facet the data by that grouping, however, and it's an entirely different story.<sup>4</sup> Both figures 9.53 and 9.54 are plotting the same data, but faceting helps to show the differences in a way that color alone could not.

---

<sup>4</sup> This dataset as a whole is the “datasaurus dozen” and is captured in the `datasauRus` package by Steph Locke and Lucy D’Agostino McGowan (<https://cran.r-project.org/package=datasauRus>). Each individual grouping of data shares the same mean and standard deviation for both the `x` and `y` values, as well as the same correlation between those.



**Figure 9.54** The datasets are revealed with facetting.

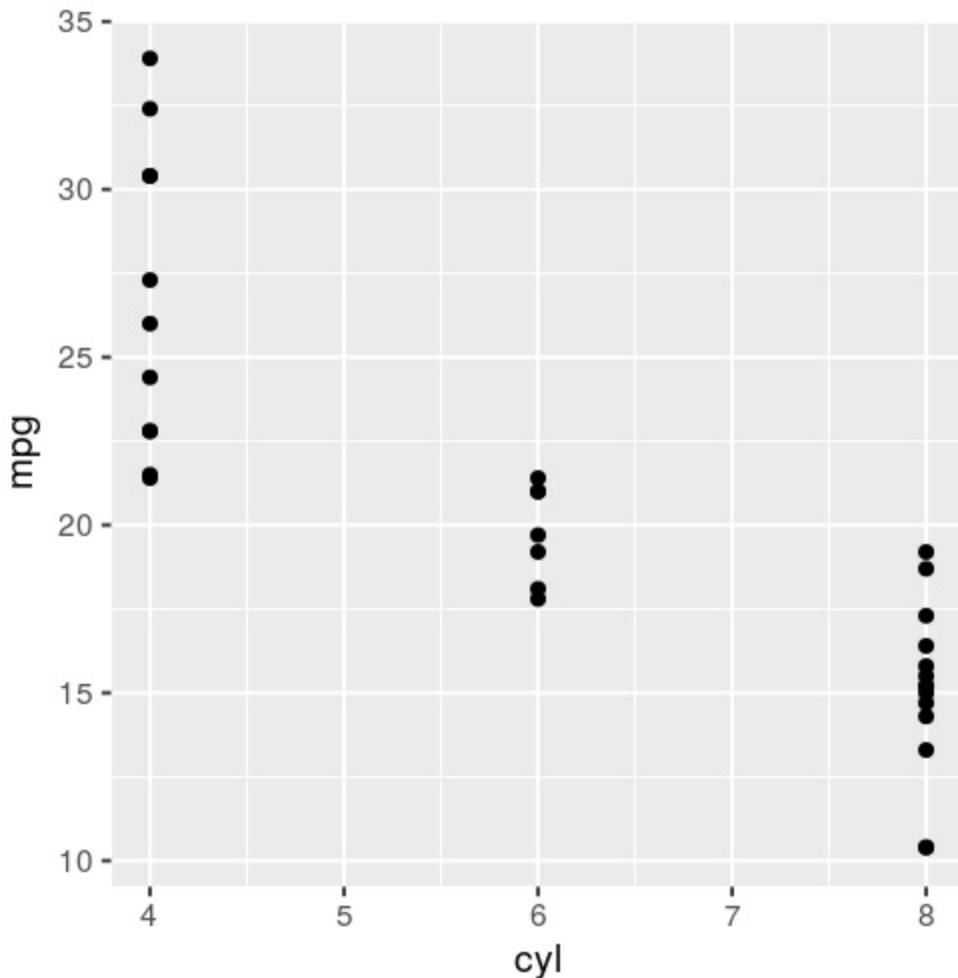
### 9.2.9 ADDITIONAL OPTIONS

All of the plots generated so far have had the same default *theme*, the grey background with gridlines. This is only the default, however, and it is entirely customizable. The `theme()` function takes a wide variety of arguments to specify everything from the background to the fonts to the way the axis labels are shown, and

so on.

Because there are so many options, various `theme_` functions exist to specify certain combinations. The default is `theme_grey()`. For example, the default plot you started with looks like [figure 9.55](#):

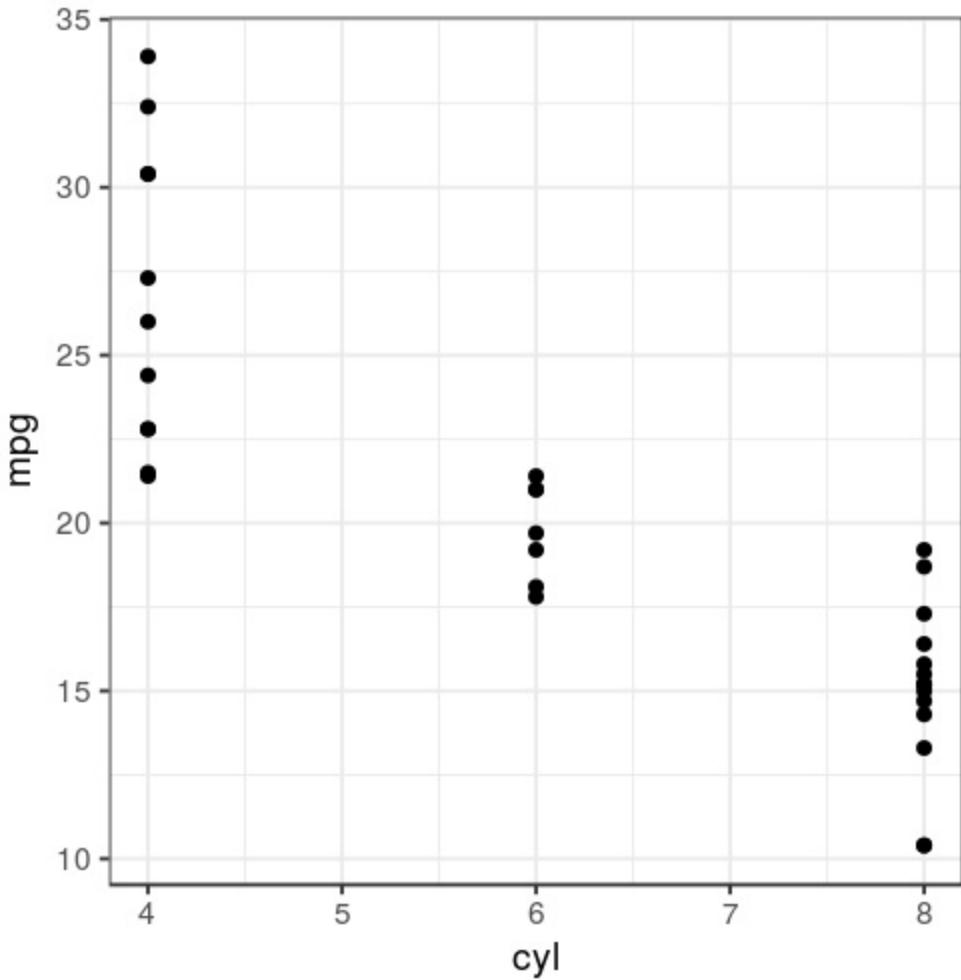
```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point()
```



**Figure 9.55** The default theme for `ggplot2` plots in action

But you can change the theme to a more black-and-white version with `theme_bw()`, as shown in [figure 9.56](#):

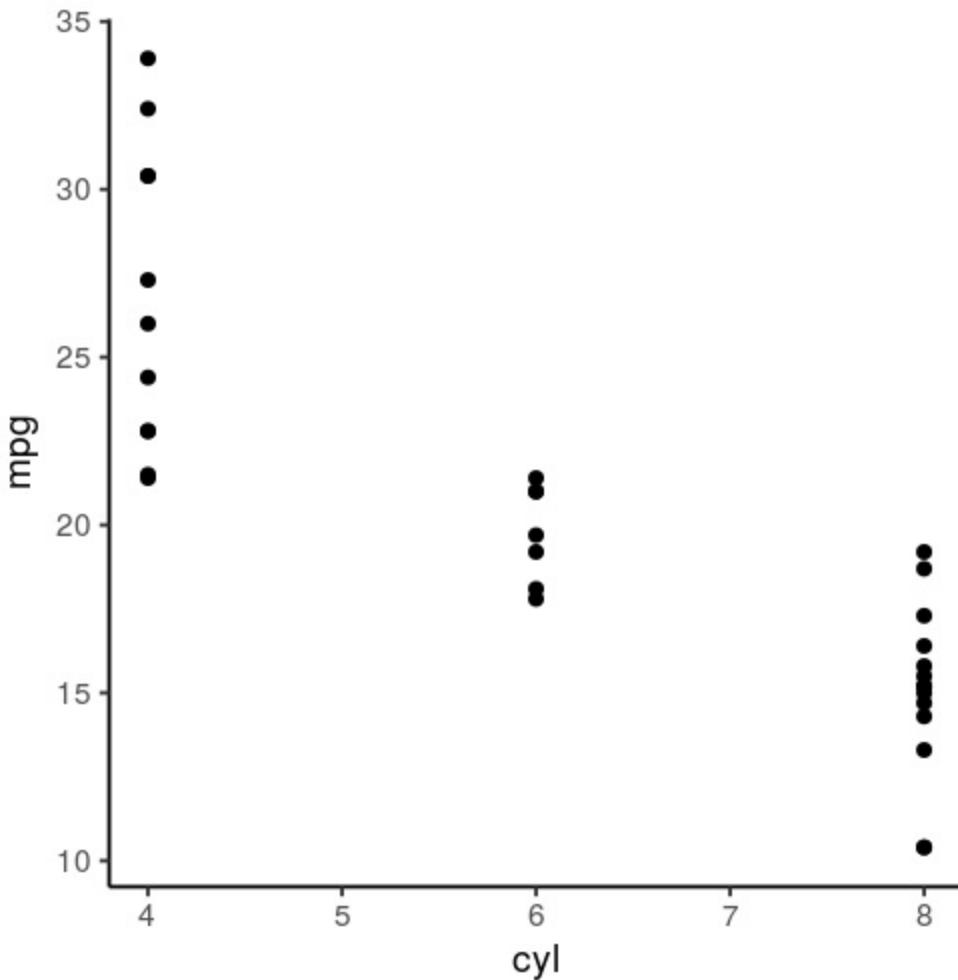
```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point() +  
  theme_bw()
```



**Figure 9.56** theme\_bw() is a bit clearer and less styled than the default.

You can remove all but the essentials with theme\_classic() (see figure 9.57):

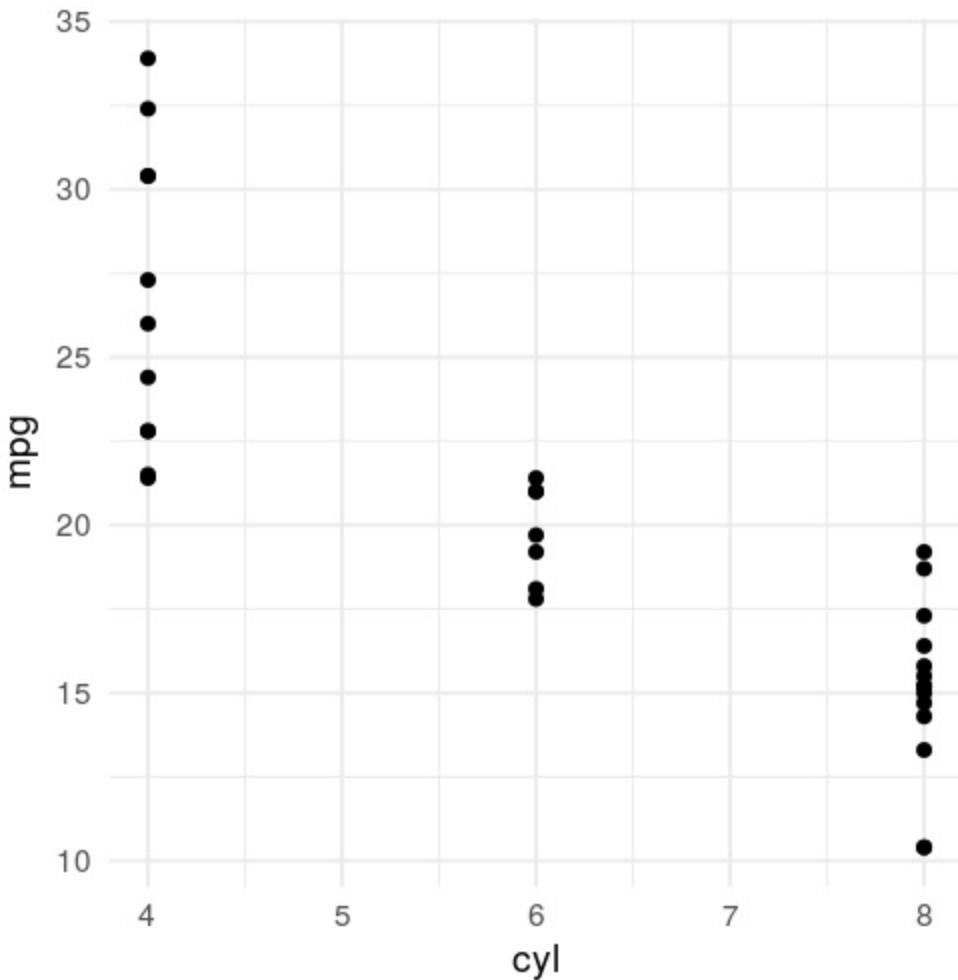
```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point() +  
  theme_classic()
```



**Figure 9.57 Just the essentials with `theme_classic()`**

Or you can remove almost everything with `theme_minimal()` (see figure 9.58):

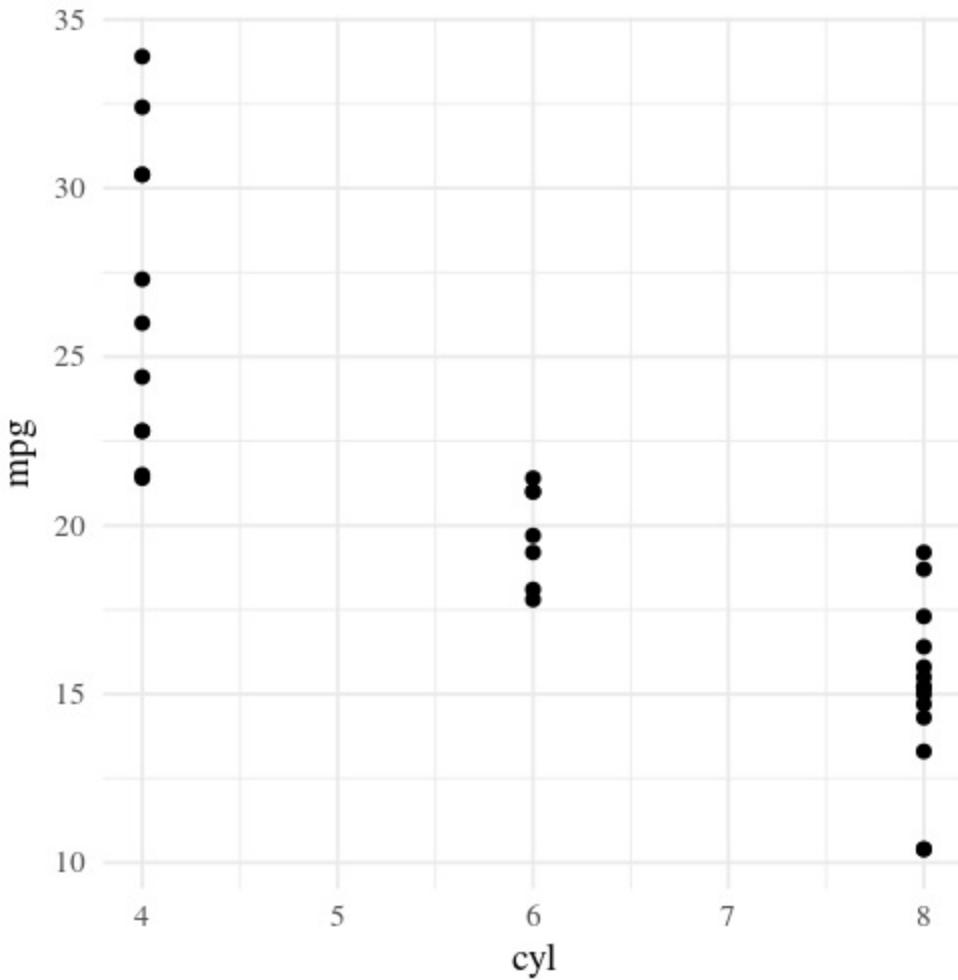
```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point() +  
  theme_minimal()
```



**Figure 9.58** This minimal plot includes very little apart from the data.

Many packages exist that contain additional theme\_ functions to extend the possibilities. You can always specify features if you also call theme(), though the syntax may require you to read through ?theme first. For example, changing the font family to serif results in figure 9.59:

```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point() +  
  theme_minimal() +  
  theme(text = element_text(family = "serif"))
```



**Figure 9.59** Changing the text family is achieved through arguments to `theme()`.

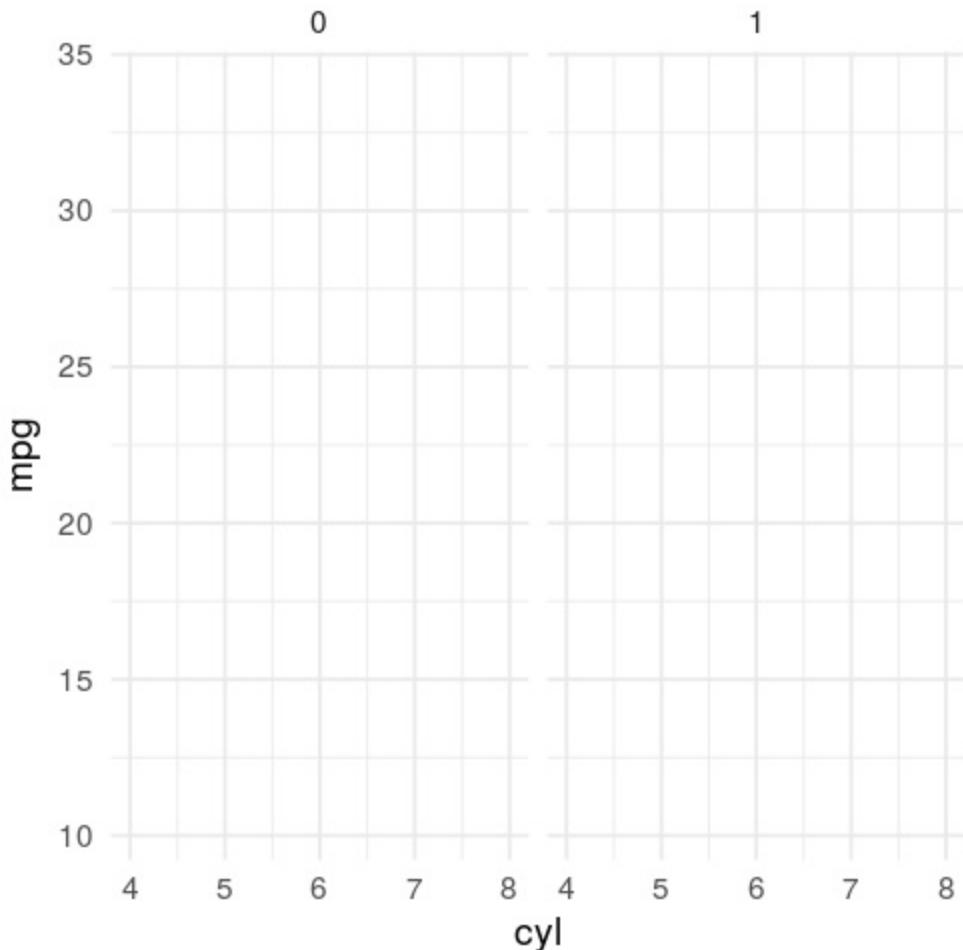
### 9.3 Plots as objects

One thing to keep in mind is that `ggplot()` returns (invisibly) an object, so you can store a plot just as you would a value. You can then modify it (with some limitations) or add to it. This becomes very handy to add different features to a basic common plot. Start with an empty plot (with aesthetics/styling and facetting defined), and save that to the variable `p`:

```
p <- ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  theme_minimal() +
  facet_wrap(~ am)
```

Note that saving the plot to `p` doesn't output anything to the Plot pane since assignment doesn't return anything (visibly). The plot is only generated once you `print()` it, which by default occurs when evaluating a `ggplot()` call on its own. If you try to print `p` (by evaluating it), you get the plot in [figure 9.60](#):

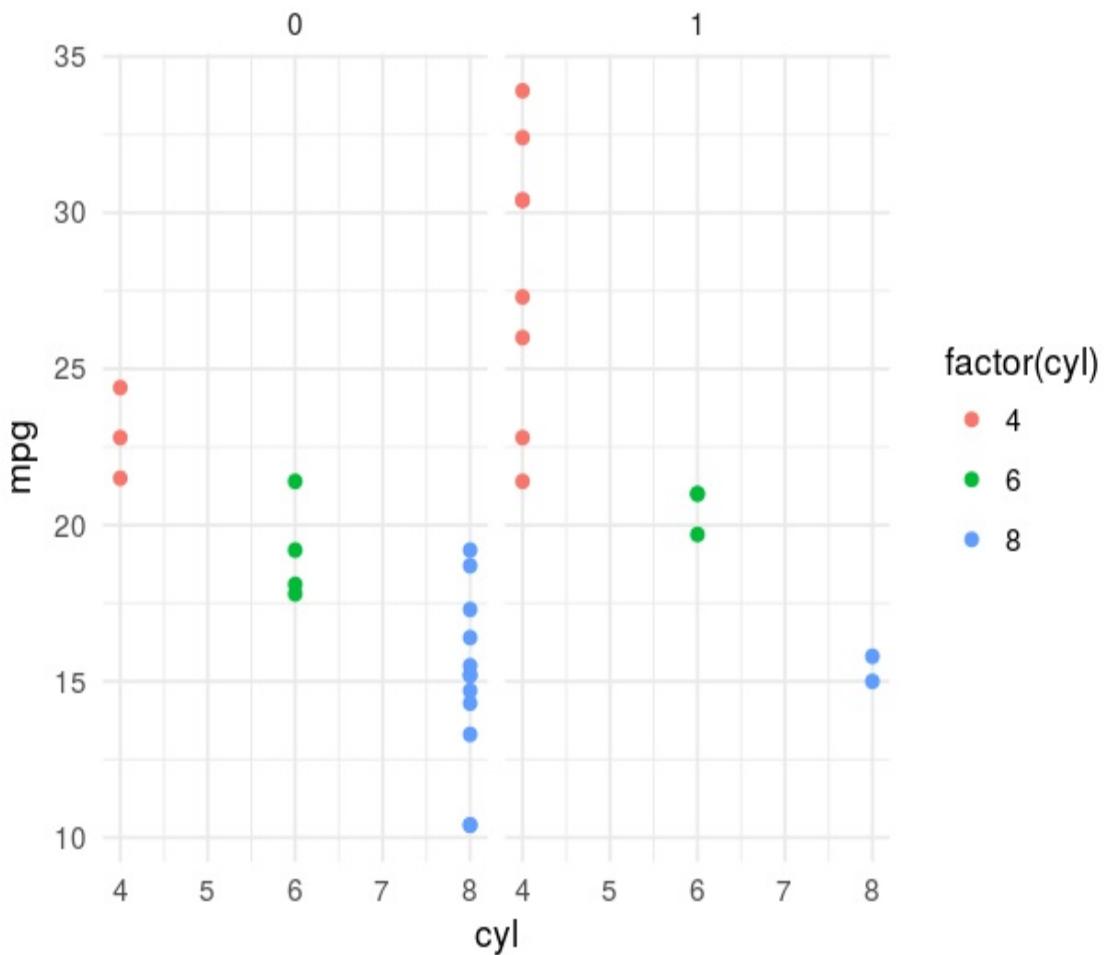
```
p
```



**Figure 9.60** A plot produced from the saved variable `p`

You can add to `p` using `+` as if you had just written the preceding code — say to add points, as in [figure 9.61](#):

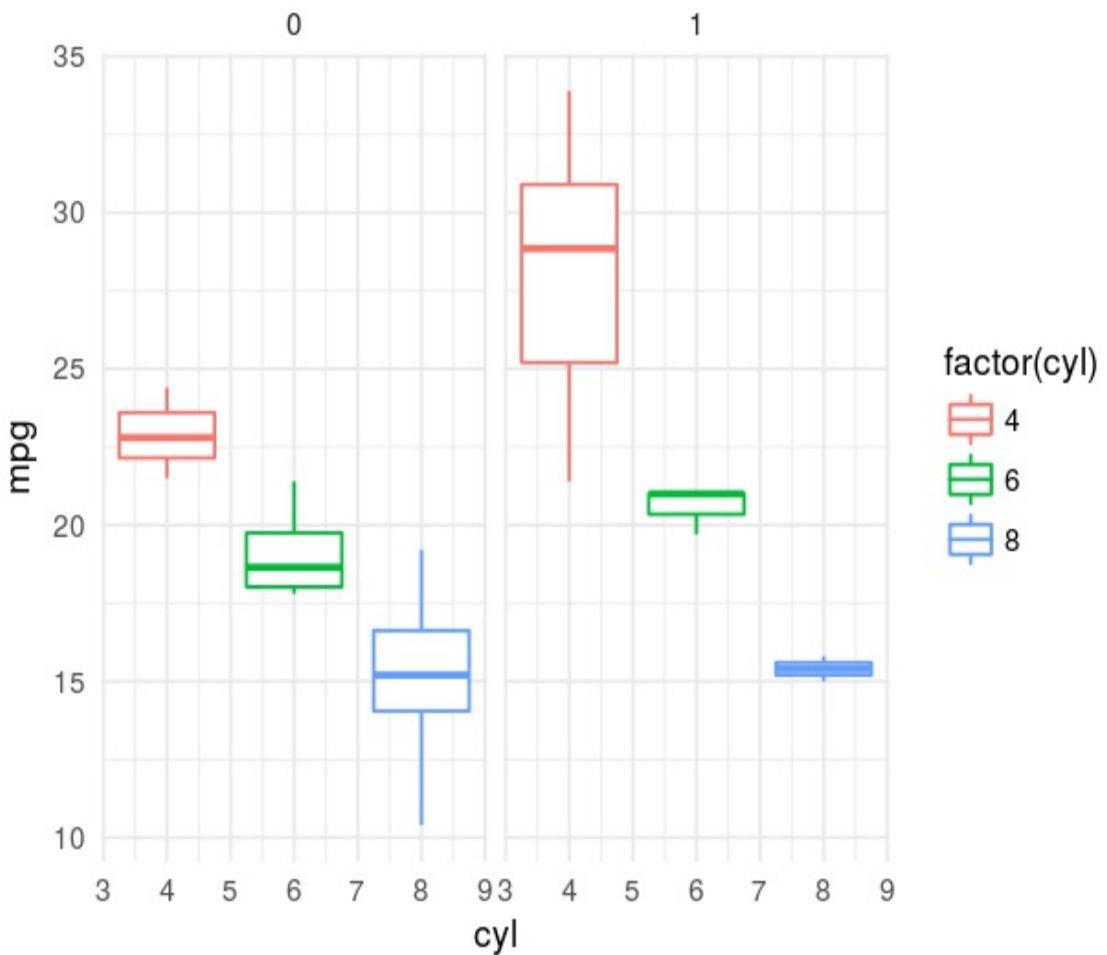
```
p + geom_point()
```



**Figure 9.61 Adding a `geom_point()` call to a variable containing a `ggplot2` plot object**

Or a box-and-whiskers plot (see figure 9.62):

```
p + geom_boxplot()
```



**Figure 9.62 Adding a `geom_boxplot()` call to a variable containing a `ggplot2` plot object**

`ggplot2` isn't a particularly new extension to R; it has a history stretching back more than a decade. Plotting has been available in R (and its predecessors) for at least a decade longer than that and hasn't changed very much in that time. There is an alternative way to create graphics in R (the base graphics method). I talk about that in appendix C.

My preference is absolutely `ggplot2`; once you're familiar with the syntax, you'll be able to throw together a basic inspection of data in mere seconds. From there, the customizable options can take you all the way to publication-quality graphics. You'll need to know how to keep them, of course.

## 9.4 Saving plots

Whichever plotting route you decide to take, at some point you'll most likely want to save plots for later use. When using RStudio, a handy Save As Image button is available in the Export dropdown at the top of the Plot pane. This provides some control over the file format and size, but it's not a reproducible step toward generating figures from an analysis.

If you're using `ggplot2`, the function `ggsave()` is very convenient, providing scripted control over the file type and size of the resulting image, with some clever processing of the input filename to determine exactly how to achieve that. If the chosen filename ends in `.jpg` or `.jpeg`, then the corresponding underlying function `jpeg()` will be used to save the plot. Similarly, if the chosen filename ends with `.png`, then the underlying function `png()` will be used. If you've saved the plot as an object, you can specify this as the `plot` argument explicitly:

```
p <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
ggsave(plot = p) ①
```

①

**Explicitly saves the plot stored in the variable `p`, regardless of whether `p` has been printed on the screen yet**

But if you leave this out, then `ggsave()` will assume you mean the last `ggplot2` plot created (even if you've produced a base plot since, and even if you haven't printed the `ggplot2` plot to the screen yet):

```
p <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
ggsave() ①
```

①

**Implicitly saves the last `ggplot2` plot generated**

Note that for the implicit `ggsave()`, the plot `p` doesn't need to have been printed to the screen at this point; the internal

components of `ggplot()` keep a running record of which plot was last generated.<sup>5</sup>

---

<sup>5</sup> You can extract this yourself with `last_plot()`.

---

## 9.5 Try it yourself

Let's bring together some of the things you've learned in the last few chapters. Choose some data (example or your own) and see if you can build an appropriate visualization with `ggplot2`. Which geometry best represents the data? A scatterplot (`geom_point()`)? A line plot (`geom_line()`)? A bar plot (`geom_bar()` or `geom_col()`)? Are the automatically chosen scales (x, y, color) appropriate? Is the story your visualization tells clearer with a different theme?

Once you're done, save your visualization to a file, and share it with me on Twitter if you like: @BSwR\_book or @carroll\_jono.

## Terminology

- *Visualization* — Visual representation of some data, most commonly involving two axes.
- *Aesthetics* — Particular aspects of a visualization — the color, shape, size, and so on — that help include more dimensions in the visualization.
- *Geometry* — The structure in which data is added to a visualization: lines, points, areas, and so forth. Multiple geometries can be added to a visualization when they help tell the story behind the data.
- *Grammar of graphics* — Leland Wilkinson's structured system for designing visualizations, involving geometries, aesthetic mappings, scales, and the way in which these connect to data.
- *Facet* — One side to the data. When you visualize data with multiple facets, you can physically separate them to better highlight similarities within a facet.

- *Theme* — The overall styling of a visualization. This is a collection of stylings for the overall look and feel of a visualization.

## ***Summary***

- Data may need restructuring in order to be visualized.
- The `ggplot2` package provides an extensive framework for plotting data.
- The features of the “grammar of graphics” are data, geometries, aesthetic mappings, and scales.
- Points, lines, and so on are added using geometries, via `geom_` functions.
- When aesthetics are provided in the `ggplot()` part of a call, they are inherited by geometries.
- `ggplot2` plots can be stored in variables and added to or saved.
- R plots follow the pen-and-paper model; once something is added to a plot it can’t be removed, and the order in which elements are added determines which will be on top of which.
- Most options are flexible and can be specified, but finding out how often requires a bit of searching. Don’t feel bad about this. A question on rotating the x-axis labels is one of the most commonly viewed in the R tag on Stack Overflow. It’s my most viewed question too.
- Factors are evenly spaced when plotted, and this is expected behavior. If you want them spaced unevenly, you can use the `scale_` to do so.
- Not everyone can see the colors in every plot the same way. Some people might be viewing on a black-and-white printout, and others may have vision issues that make distinguishing certain colors difficult. Take care with your visualization choices to not exclude anyone.

# 10

## *Doing more with your data with extensions*

---

### **This chapter covers**

- Writing your own packages
- Analyzing your code (benchmarking/profiling)
- Making code more efficient
- Figuring out where to go next
- Communicating your work

If all has gone well, you now have the tools you need to take command of your data and extract insights from it in a reproducible, readable, and understandable way. That's not the end of the line for learning about R, though. As you gain more experience, you'll find certain parts that you wish were better. Maybe your code runs too slowly to be useful, maybe you need more rigor in the functions you've written, or maybe you've made something awesome and want to share it with the world. This chapter covers how to do these things should you decide you want to. There's a lot to learn, so I'll cover the basics and hopefully give you enough of a head start to track down the remaining pieces of your particular puzzle.

## ***10.1 Writing your own packages***

Whether you have a single function or a whole directory full of files containing functions, it's often a good idea to solidify these into a package. Not only does this allow some formal testing of your function(s), but it also provides a very good mechanism for documentation, and at the end of it you have something you can more easily share with others.

I write many of my analyses as packages, even very small ones. This provides a way to include the data, perform testing on the functions I create, and wrap it all up in a neat, tidy framework that can be shared or stored.

At its simplest, a minimal R package directory (named as the name of your package) will contain the following:

- DESCRIPTION file
- NAMESPACE file
- subdirectory named R

And that's it.

### ***10.1.1 CREATING A MINIMAL PACKAGE***

The easiest way to create a minimal package is to use the `devtools` package and its function `create()`, passing an (empty) package directory location in as a string. This performs several steps for you:

- Creates a DESCRIPTION file with valid defaults
- Creates a NAMESPACE file with a catch-all default
- Adds an empty R subdirectory
- Creates a project file, `yourPackage.Rproj` (covered in section 1.4)
- Creates a `.gitignore` file, already set up to ignore certain files
- Creates `.Rbuildignore`, already set up to ignore project files

This takes care of most of what's required to get a package working. If you open the yourPackage.Rproj file (which `create()` just created) in RStudio (File > Open Project), you'll notice a new Build tab in one of the panes. This has a button called Build & Reload (keyboard shortcut: Ctrl-Shift-B) that will trigger the build process and create your package from the preceding files.

If this goes well, you'll see something like this:

```
No man pages found in package 'yourPackage'
* installing to library '/home/user/R/x86_64-pc-linux-gnu-
library/3.4'
* installing *source* package 'yourPackage' ...
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (yourPackage)
```

That `* DONE (yourPackage)` is what you see when a package has successfully been installed, and that's the case here. You can now run `library(yourPackage)`, and it should work.

That's not much use yet, of course, because the package doesn't do anything. You need to provide some functions for the package to provide back. If you now create a file `example_function.R` in the R directory and write a function in it, such as

```
makeMeSmile <- function() {
  message("I can develop packages now!")
}
```

and then rebuild the package, you'll again see the `library(yourPackage)` line in the Console. If you start typing the name of the function, it will hopefully autocomplete. Evaluating it with no input (it doesn't use any yet), you should see the following:

```
makeMeSmile()
# I can develop packages now!
```

You can return to the script and improve it; let's make it a bit

more personal.

Add an argument to the function — perhaps one that will take your name as a string. Now update the function body to use this information in the message.

An example might be as follows:

```
makeMeSmile <- function(me = "I") { ①
  message(me, " can develop packages now!") ②
}
```

①

**Providing a default argument is often a good idea.**

②

**message() takes any number of arguments and paste0()s them together before writing to Console.**

Save the file, and rebuild the package. Now try your function in the Console with and without the argument:

```
makeMeSmile("You")
# You can develop packages now!
```

## 10.1.2 DOCUMENTATION

You should add some documentation to this function, for your own benefit and the benefit of anyone using it (even if that is just you). This will be processed into a help file just like any other function has, so you will be able to access it with `?makeMeSmile`. Another extension package to R is the `roxygen2` framework. This provides a mechanism by which you can add documentation of a function in close proximity to the function, which helps keep the two in mind at the same time. This requires the `roxygen2` package, which you'll need to install:

```
# install.packages("roxygen2")
```

You don't actually need to use any of the `roxygen2` functions, so

there's no need to attach these using `library()`; just having the package installed is enough.

The easiest way to document the functions you're writing is with RStudio. Place your cursor anywhere within your function definition, and click `Code > Insert Roxygen Skeleton`. When you do that, possibly after installing the latest `roxygen` package, the following will appear just above your function like magic:

```
#' Title      ①
#'
#' @param me    ②
#'
#' @return     ③
#' @export      ④
#'
#' @examples
makeMeSmile <- function(me = "I") {
  message(me, " can develop packages now!")
}
```

①

A title that will appear at the top of the help file for your function

②

Each argument (also known as a parameter) needs to be described with an `@param` tag, the name of the variable, and a description. Clicking the `Insert Roxygen Skeleton` button autopopulates this with existing argument names.

③

Describes what object the function will return, if anything

④

If you want the function to be available to a user after they run `library()`, it must be exported, in which case it appears in the `NAMESPACE` file.

These lines all begin with # so they're treated as comments by the R interpreter, but when used along with an apostrophe as the next character ('), the roxygen2 package will treat them as special documentation code. roxygen2 uses a system of tags that begin with @ to build up the help file for your function. By default, the first line of the roxygen block denotes the title to be used in your help file. The next few lines can contain the extended description that will appear just below the title as a Description section.

@param is used to describe each parameter and creates the Arguments section in the help file; you'll need one per argument in your function, each starting with @param. It should start with the name of the argument (in the example, mine is me), followed by anything the user will need to know about it (default value, restrictions on which values it can take, and so forth). Keep in mind that these won't update if you change the function arguments, but having them in such close proximity will hopefully remind you to keep them in sync.

@return describes what the function will return and creates the Value section in the help file. In this case, it's nothing (NULL), so you can write that.

@export is a flag that indicates that the NAMESPACE file should be updated to explicitly include the function. Exporting a function means it is explicitly available to the user (it will show up in autocompletes and can be used easily). Not including this flag indicates that the function is typically for internal use only (say, it's needed by another of your functions, but isn't intended to be used by a user). Not marking a function as exported doesn't mean a user *can't* use it, it just means they'll need to override the namespace search to do so (using yourPackage:::internalFunction — note the three colons).

@examples is where you show users how to use the function. These show up at the bottom of the help file and should cover most common uses of arguments and corner cases to watch out for. Too often, functions have too few examples or the examples

don't work, so be kind to your users and show them how anything difficult to use works.

Another tag that's good to add is `@details`. It adds a Details section to the help file and is where you describe the ins and outs of the function.

Here's an example of how this documentation might look for the example function:

```
#' Reassure A User That They Are Doing Well
#'
#' Look at that, they're building a documented function
already!
#'
#' @param me the user to reassure
#'
#' @return NULL (invisibly). Used for the side-effect of
generating a
#'   \code{message}.
#' @export
#'
#' @examples
#' makeMeSmile("You")
makeMeSmile <- function(me = "I") {
  message(me, " can develop packages now!")
}
```

Code can be specified (and accordingly styled) using the `\code{}` syntax.

A short but solid example in a real-world package is the `roxygen2` documentation for the `tribble()` function from the `tibble` package (as of version 1.4.1.):

```
#' Row-wise tibble creation ①
#'
#' Create [tibble]s using an easier to read row-by-row
layout. ②
#' This is useful for small tables of data where
readability is
#' important. Please see \link{tibble-package} for a
general introduction.
#'
#' `frame_data()` is an older name for `tribble()`. It will
eventually ③
```

```

#' be phased out.
#
#' @param ... Arguments specifying the structure of a
`tibble`. ④
#' Variable names should be formulas, and may only appear
before the data.
#' @return A [tibble]. ⑤
#' @export ⑥
#' @examples ⑦
#' tribble(
#'   ~colA, ~colB,
#'   "a", 1,
#'   "b", 2,
#'   "c", 3
#' )
#'
#' # tribble will create a list column if the value in any
cell is
#' # not a scalar
#' tribble(
#'   ~x, ~y,
#'   "a", 1:3,
#'   "b", 4:6
#' )
tribble <- function(...) { ⑧
  data <- extract_frame_data_from_dots(...)
  turn_frame_data_into_tibble(data$frame_names,
data$frame_rest)
}

```

①

**The title that appears at the top of the help file**

②

**Description field**

③

**Additional Details field**

④

**This function only takes the catch-all ... argument, and what happens to those values is documented.**

(5)

**The return object is described by its class.**

(6)

**This function is exported so it's available to the user.**

(7)

**Useful examples to help users understand how to use the function**

(8)

**Finally, the short code is provided — here, there's more documentation than code. Remember to put sufficient time into your own documentation.**

This produces the help file `?tribble`, which looks like [figure 10.1](#).

tribble {tibble}

R Documentation

## Row-wise tibble creation

### Description

Create [tibbles](#) using an easier to read row-by-row layout. This is useful for small tables of data where readability is important. Please see [tibble-package](#) for a general introduction.

### Usage

```
tribble(...)
```

### Arguments

... Arguments specifying the structure of a tibble. Variable names should be formulas, and may only appear before the data.

### Details

`frame_data()` is an older name for `tribble()`. It will eventually be phased out.

### Value

A [tibble](#).

### Examples

```
tribble(
  ~colA, ~colB,
  "a", 1,
  "b", 2,
  "c", 3
)

# tribble will create a list column if the value in any cell is
# not a scalar
tribble(
  ~x, ~y,
  "a", 1:3,
  "b", 4:6
)
```

---

[Package `tibble` version 1.4.1 [Index](#)]

**Figure 10.1** The help file `?tribble` produced by the preceding `roxygen2` code

If you rebuild and try `?makeMeSmile`, you may receive a discouraging sign:

No documentation for 'makeMeSmile' in specified packages  
and libraries:  
you could try '??makeMeSmile'

For whatever reason, the default for some versions of RStudio is set to not document everything on every build. You can change this in the settings. In Build > Configure Build Tools is a Generate Documentation with Roxygen check box. This should be checked. Furthermore, clicking the Configure button brings up the Roxygen Options menu, the latter part of which has an Automatically Roxygenize When Running section; check Build & Reload. Click the OK buttons until you get back to the editor pane. Now when you re-build you should see

```
Writing makeMeSmile.Rd
```

If you try `?makeMeSmile` now, you should see the help page for the function, as shown in figure 10.2.

`makeMeSmile {yourPackage}` R Documentation

## Reassure A User That They Are Doing Well

### Description

Look at that, they're building a documented function already!

### Usage

```
makeMeSmile(me = "I")
```

### Arguments

`me` the user to reassure

### Value

`NULL` (invisibly). Used for the side-effect of generating a message.

### Examples

```
makeMeSmile("You")
```

---

[Package `yourPackage` version 0.0.0.9000 [Index](#)]

**Figure 10.2** The help page for the function you just created

This is the core of building a package: write and document

functions, then build. Lather, rinse, repeat.

## 10.2 Analyzing your package

Making a package is one thing, but are you sure it does what you think it does? Does it do that well? One of the strengths of the community-driven nature of the R language is that extensive testing is often performed on packages before they're adopted by the wider community. This instills a level of trust that the package `calculate_x` will actually calculate `x` with some validity. There are several ways to make a package reliable and trustworthy, and I cover some of those briefly here.

### 10.2.1 UNIT TESTING

Are you certain that your functions do what you expect them to do? For any input? An old rule in programming is to never trust that the user will always use sensible input.

How do you know that the latest change you made to your function didn't break the way it works? Thankfully, people have considered this already, and you can enlist another extension package to ensure that your assumptions hold true. The `testthat` package provides a mechanism to add *unit tests* to a package:

```
# install.packages("testthat")
library(testthat)
```

**DEFINITION** A *unit test* is an evaluation that confirms whether or not a particular assumption made by code is true. This is performed at the smallest level possible (a *unit* of code); although it may be true that a program as a whole runs, unit testing performed at the function level ensures that a given input produces a given output (or error, warning, and so on).

These tests can be evaluated regularly to check assumptions between what a function produces and what is expected.

Suppose a small part of the codebase is a function that adds two numbers:

```
add <- function(x, y) {
  return(x + y)
}
```

You can test that it produces what you expect with different inputs:

```
test_that("normal addition rules apply", {
  expect_equal(add(1L, 1L), 2L) ②
  expect_equal(add(-1, 1), 0) ③
  expect_error(add(1, "1")) ④
  expect_error(add(1))
  expect_type(add(1, 1), "double") ⑥
  expect_type(add(1L, 1L), "integer") ⑥
})
```

①

**Provides some context to what's being tested**

②

**Expectations take the form `expect_x` and test that `x` is true.**

③

**You expect that the result of the first argument is equal to the second argument.**

④

**Tests that code doesn't work when it's expected to fail.**

⑥

**Tests that you receive the type of data expected**

The result of calls to `test_that()` should be error-free and invisible, so don't worry if you see nothing. You know you've made a mistake when that's not true:

```

test_that("adding two differently sized data.frames", {
  expect_silent(add(mtcars, iris))
})

#> Error: Test failed: 'adding two differently sized
data.frames'
* '+' only defined for equally-sized data frames
<traceback truncated>

```

Unit tests are a great way to really poke at a function and see if it does what you think it does. For example, if the user provides a `data.frame` and a number, what will the `add()` function return? Test it with an assumption — that it should silently work:

```

test_that("adding a data.frame to a number", {
  expect_silent(add(mtcars, 1))
})

```

This returns nothing because the evaluation is correct; `add()` (because it's only `+`) can deal with this scenario just fine:

```

head(
  add(mtcars, 1)
)
#>                               mpg cyl disp  hp drat    wt  qsec vs
am gear carb
#> Mazda RX4           22.0   7 161 111 4.90 3.620 17.46  1
2   5   5
#> Mazda RX4 Wag       22.0   7 161 111 4.90 3.875 18.02  1
2   5   5
#> Datsun 710          23.8   5 109  94 4.85 3.320 19.61  2
2   5   2
#> Hornet 4 Drive      22.4   7 259 111 4.08 4.215 20.44  2
1   4   2
#> Hornet Sportabout   19.7   9 361 176 4.15 4.440 18.02  1
1   4   3
#> Valiant              19.1   7 226 106 3.76 4.460 21.22  2
1   4   2

```

Every value is increased by 1, compared to the input:

```

head(mtcars)
#>                               mpg cyl disp  hp drat    wt  qsec vs
am gear carb
#> Mazda RX4           21.0   6 160 110 3.90 2.620 16.46  0
1   4   4

```

```
#> Mazda RX4 Wag      21.0   6 160 110 3.90 2.875 17.02  0
1     4     4
#> Datsun 710        22.8   4 108  93 3.85 2.320 18.61  1
1     4     1
#> Hornet 4 Drive    21.4   6 258 110 3.08 3.215 19.44  1
0     3     1
#> Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0
0     3     2
#> Valiant           18.1   6 225 105 2.76 3.460 20.22  1
0     3     1
```

What if you use two *compatible* data.frames?

```
test_that("adding two data.frames", {
  expect_silent(add(mtcars, mtcars))
})
```

Again, + is made to deal with this:

```
head(
  add(mtcars, mtcars)
)
#>                               mpg cyl disp  hp drat    wt  qsec vs
am gear carb
#> Mazda RX4        42.0  12 320 220 7.80 5.24 32.92  0
2     8     8
#> Mazda RX4 Wag   42.0  12 320 220 7.80 5.75 34.04  0
2     8     8
#> Datsun 710       45.6   8 216 186 7.70 4.64 37.22  2
2     8     2
#> Hornet 4 Drive   42.8  12 516 220 6.16 6.43 38.88  2
0     6     2
#> Hornet Sportabout 37.4  16 720 350 6.30 6.88 34.04  0
0     6     4
#> Valiant          36.2  12 450 210 5.52 6.92 40.44  2
0     6     2
```

You can include tests in a package so that they can be regularly evaluated when you make changes. The easiest way to accomplish this is with `devtools::use_testthat()`. A good rule of thumb is that any time you discover a *bug* in your code (something happened that you didn't expect — a wrong calculation, a broken assumption, an unexpected result on a rarely used input), you should write a unit test to check for it. To be truly proactive, you can even write the test (knowing what you

expect the output to be, given an input, regardless of the implementation) *before* writing the actual function/implementation.

This is an extensive topic of its own, so I'll direct you to the "Testing" section of the book *R Packages* (O'Reilly, 2015) by the author of the `testthat` package, Hadley Wickham, for more information (<http://r-pkgs.had.co.nz/tests.html>).

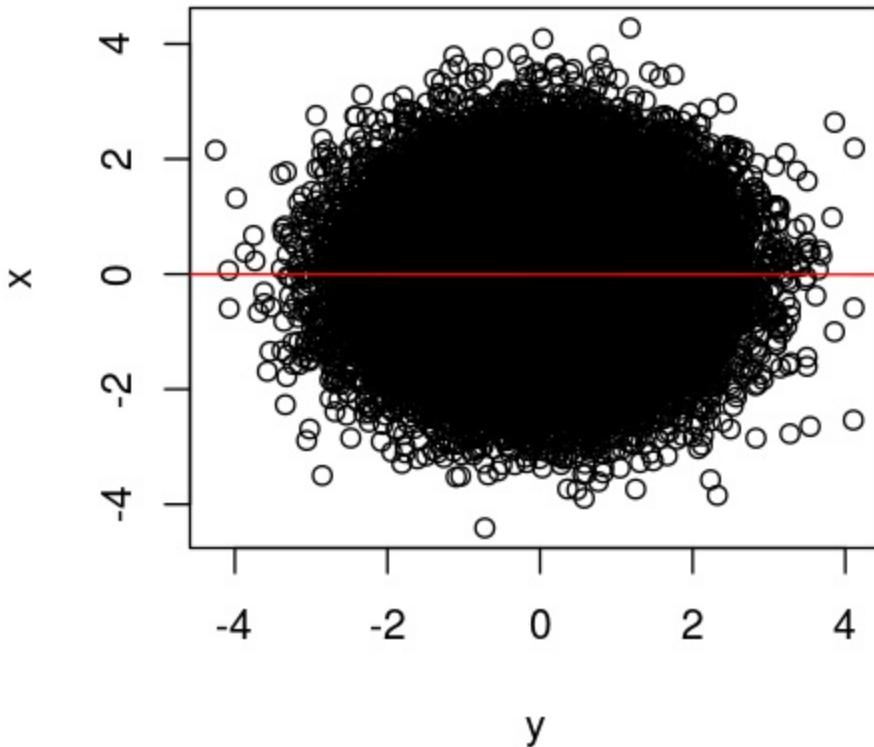
## 10.2.2 PROFILING

Code in a package may be documented and tested, but it can also run slower than a week of Sundays. We've been through best practices for making code clear and efficient (section 8.1.2) but what should you do if a function is just taking too long to evaluate?

Enter *profiling*, a way to analyze the evaluation of your code at the R level and see which parts are taking the longest to complete. RStudio comes with this functionality built in, thanks to the `profvis` package (<https://rstudio.github.io/profvis>). Here's some code that performs several operations: generates some data, calculates a linear model fit between two columns, and plots that line on top of the data (producing figure 10.3, taken from the help page for `profvis`). The overall timing of evaluating these isn't impressive:

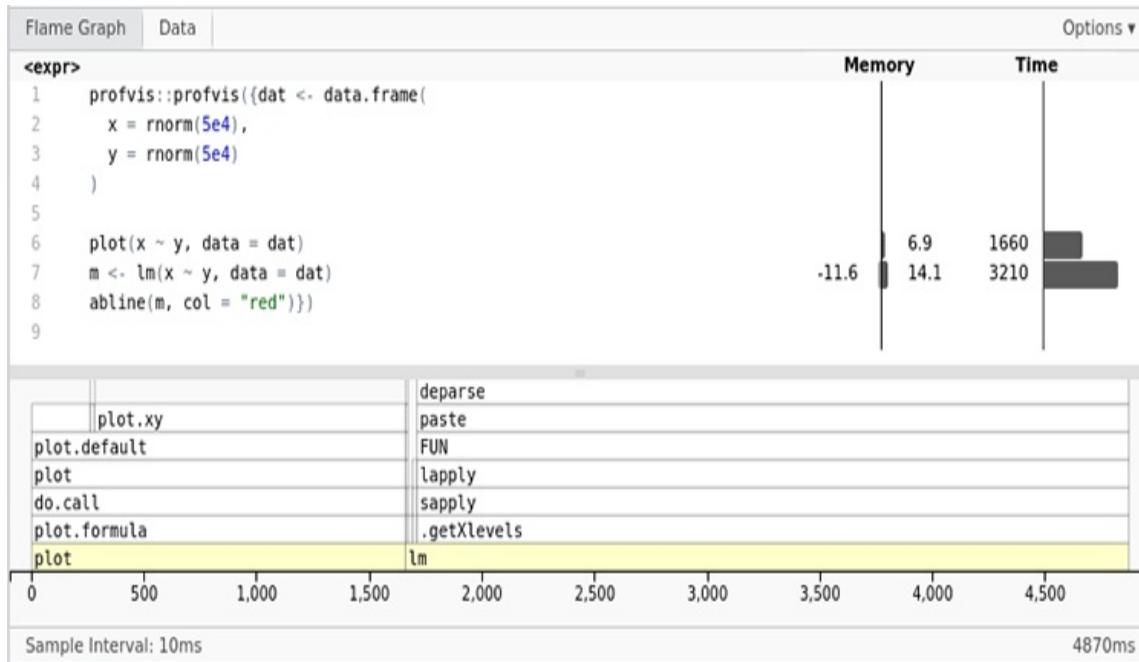
```
dat <- data.frame(
  x = rnorm(5e4),
  y = rnorm(5e4)
)

plot(x ~ y, data = dat)
m <- lm(x ~ y, data = dat)
abline(m, col = "red")
```



**Figure 10.3 Example from `?profvis`**

In this example, or in your own code, it may not be obvious which line is causing the delay. Is it the data generation? The fitting? The plotting? You can run the profiler over this by highlighting the evaluation lines and clicking Profile > Profile Selected Lines. This evaluates the code, keeping track of which parts run how long (including the internal components of each call). You're provided with a timing breakdown in the form of a chart, as shown in figure 10.4, which compares the memory usage and timing of various components of the call. The call to `plot()` takes around one third of the total time (1660 ms out of the total 4870 ms), with the vast majority of the remaining time (3210 ms) being due to the `lm()` fitting. The `lm()` step also used the most memory (14.1 MB, though it did result in deallocating 11.6 MB also as part of R's memory cleanup routines).



**Figure 10.4** profvis output

Although you may have guessed that the fitting would be the slow step, it's not always obvious which step is taking up all the processing time. For more details, see the profvis site at <https://rstudio.github.io/profvis>.

## 10.3 What to do next?

Now that you have the basic and intermediate tools at your disposal, it's time to let you loose on your own projects. This is where you'll need to decide for yourself which direction to take — or at least start discovering the more advanced tools you'll need to use. Covering every possible data analysis tool would be an insurmountable task; entire books are written on very small aspects of data analysis performed using R. Instead, I'll point you in the direction of several common analysis tasks so you have some vocabulary to begin your search.

### 10.3.1 REGRESSION

One of the most straightforward (in terms of approachability, not interpretation) analyses commonly undertaken is a *regression*.

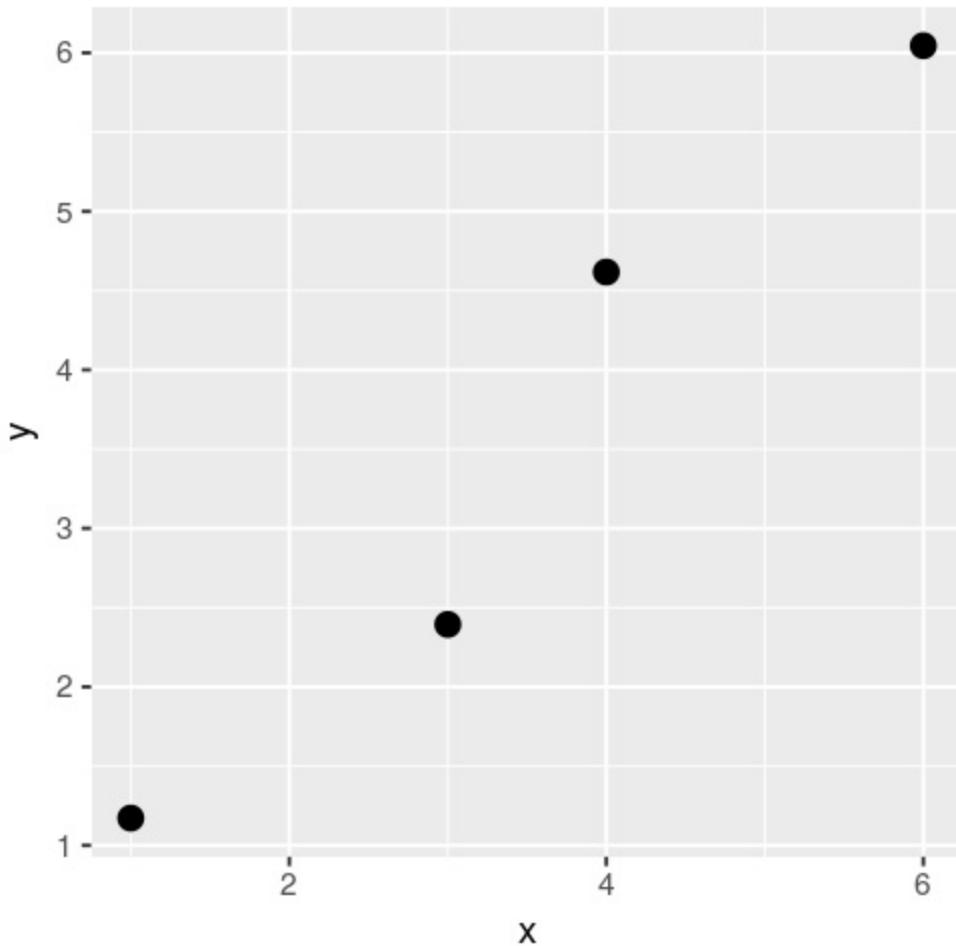
In terms of R, this involves using functions to determine some relationship between data values. The simplest version of this is a *linear regression*, in which you fit a straight line between two sets of values. This is often referred to as a *linear model*, and therefore the R function that performs this task is named `lm()`.

Say you have some measured values `y` that were observed at values `x`:

```
x <- c(1, 3, 4, 6)
y <- c(1.172, 2.394, 4.617, 6.045)
```

These are plotted in figure 10.5:

```
# install.packages("ggplot2")
library(ggplot2)
z <- data.frame(x, y)
ggplot(z, aes(x, y)) + geom_point(size = 3) + coord_equal()
```



**Figure 10.5 Values with a linear-like relationship**

You can fit a linear model to these points using `lm()`, which takes a formula (optionally referring to columns in a `data.frame` provided as a `data` argument):

```
linear_model <- lm(y ~ x)
```

The resulting object has its own class `lm` and contains many components that define the model. This class has its own `print` method, so you can produce output from this object by evaluating it:

```
linear_model  
#>  
#> Call:  
#> lm(formula = y ~ x)  
#>
```

```
#> Coefficients:
#> (Intercept)           x
#> -0.02215      1.02262
```

You can extract more information by requesting a summary of the model:

```
summary(linear_model)
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Residuals:
#>    1     2     3     4
#> 0.17154 -0.65169  0.54869 -0.06854
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.02215   0.67306  -0.033  0.9767
#> x            1.02262   0.17096   5.982  0.0268 *
#> ---
#> Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
#> ' '
#>
#> Residual standard error: 0.6164 on 2 degrees of freedom
#> Multiple R-squared:  0.9471, Adjusted R-squared:  0.9206
#> F-statistic: 35.78 on 1 and 2 DF,  p-value: 0.02683
```

The slope and intercept are the two coefficients calculated, and these determine the equation describing the line of best fit, or linear model, as  $y \sim m*x + c$  for slope  $m$  and intercept  $c$ . More importantly, you can predict what  $y$  values this model would produce at any arbitrary values of  $x$ :

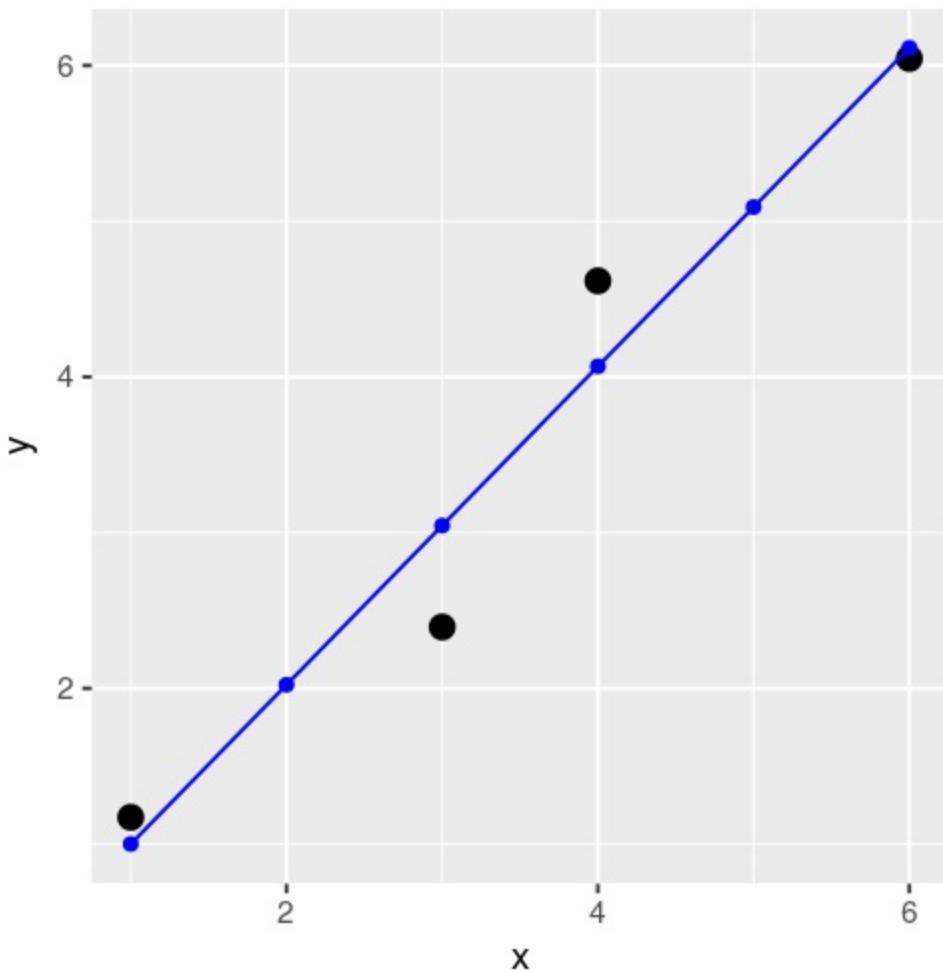
```
prediction_x <- 1:6
prediction_y <- predict(linear_model, newdata =
data.frame(x = prediction_x))

predicted <- data.frame(x = prediction_x, y = prediction_y)
predicted
#>   x     y
#> 1 1 1.000462
#> 2 2 2.023077
#> 3 3 3.045692
#> 4 4 4.068308
#> 5 5 5.090923
```

```
#> 6 6 6.113538
```

If you plot these predicted  $y$  values at each of these  $x$  values on top of the data points, they follow an exactly straight line; this is the linear model fitted to the data. These are shown in [figure 10.6](#):

```
ggplot(z, aes(x, y)) +
  geom_point(size = 3) +
  geom_point(data = predicted, aes(x, y), col = "blue") +
  geom_line(data = predicted, aes(x, y), col = "blue") +
  coord_equal()
```



**Figure 10.6 Values with a linear-like relationship and line of best fit**

Not every paired vector of  $x$  and  $y$  values can be well described by a linear model, but that doesn't stop R or an analyst from doing so. Model selection and interpretation are extensive topics in and

of themselves, and I recommend you consult some more specific statistics texts before becoming too attached to any particular model. R places many regression functions at your disposal, including non-linear regression, generalized linear models (GLMs), general additive models (GAMs), and many, many more. R began as a statistical language, so this is certainly a topic in which it is well supported.

### 10.3.2 CLUSTERING

Sometimes the effect you're interested in isn't described by an equation relating variables. Rather, you may be interested in *clusters* within your data — placing the data into two or more groups based on a combination of variables that makes them more similar to others within a group than those in other groups. For example, you may have some points that belong to two somewhat distinct groups, even if they appear as one continuous group, such as those in figure 10.7:

```
# install.packages("dplyr")
library(dplyr)

sdxy <- 0.9
set.seed(2)
spread_data <- sample_frac( ①
  data.frame(x = c(
    3 + rnorm(100, sd = sdxy),
    6 + rnorm(100, sd = sdxy)
  )) %>% mutate(
    y = x + rnorm(100, sd = sdxy),
    source = rep(c("A", "B"), each = 100)
  ),
  1
)
head(spread_data, 10) ②
#>          x      y source
#> 76  2.183601 3.628281     A
#> 18  3.032226 1.840193     A
#> 75  2.951650 3.177482     A
#> 45  3.560245 4.261439     A
#> 160 6.600770 6.446583     B
#> 28  2.463098 3.267834     A
#> 6   3.119178 3.464633     A
```

```
#> 44 4.713193 5.207872
#> 148 6.494918 6.748288
#> 122 6.124973 6.768940
```

```
A
B
B
```

```
ggplot(spread_data, aes(x, y)) + geom_point(size = 3)
```

①

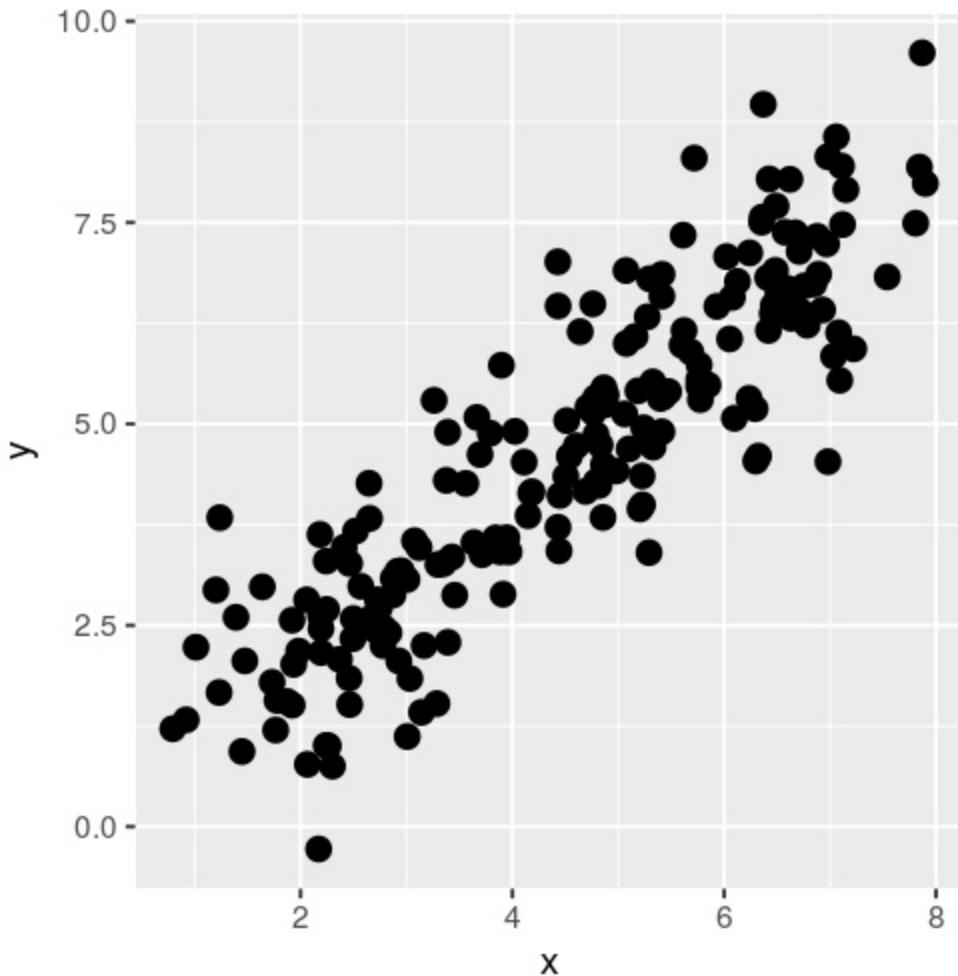
**Don't worry too much about the code used to generate these points.**

②

**Shows the first 10 rows of the data, rather than the default of 6**

You can use the *k-means* clustering algorithm to assign each point to one of two clusters (the number of clusters needs to be specified ahead of time):

```
km <- kmeans(spread_data[, c("x", "y")], centers = 2)
str(km)
#> List of 9
#> $ cluster      : Named int [1:200] 1 1 1 1 2 1 1 2 2 2
...
#>   ...- attr(*, "names")= chr [1:200] "76" "18" "75" "45"
...
#> $ centers      : num [1:2, 1:2] 2.78 5.88 2.77 6.12
#>   ...- attr(*, "dimnames")=List of 2
#>     ...$ : chr [1:2] "1" "2"
#>     ...$ : chr [1:2] "x" "y"
#> $ totss        : num 1472
#> $ withinss     : num [1:2] 181 262
#> $ tot.withinss: num 442
#> $ betweenss    : num 1029
#> $ size         : int [1:2] 89 111
#> $ iter         : int 1
#> $ ifault       : int 0
#> - attr(*, "class")= chr "kmeans"
```



**Figure 10.7** Grouped points. These were produced from two distinct groups but are spread out enough that they appear like one group.

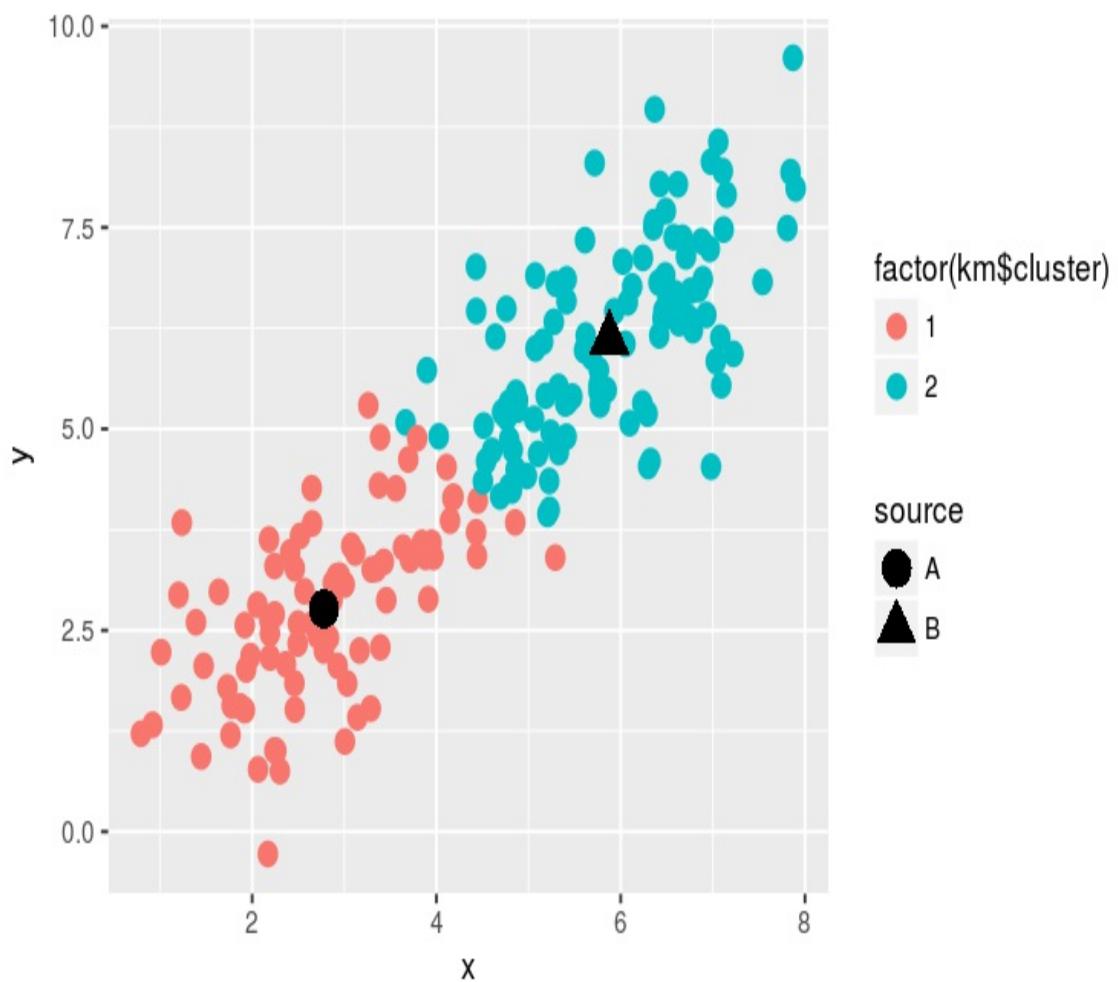
The `kmeans()` function performs this task and returns a list containing the clustering information. You can color the points according to this information, which makes the clustering much more apparent, as shown in figure 10.8:

```
ggplot(spread_data, aes(x, y)) + ①
  geom_point(size = 3, aes(col = factor(km$cluster))) +
  geom_point(data = data.frame(
    km$centers,
    z = unique(km$cluster)
  ), aes(x, y, shape = c("A", "B")), size = 5) +
  guides(shape = guide_legend(title = "source"))
```

①

Don't worry too much about the code used to generate the clusters.

**NOTE TO PRINT BOOK READERS** As mentioned in the previous chapter, many graphics in this book are meant to be viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to <https://www.manning.com/freebook> and follow the instructions to complete your pBook registration.



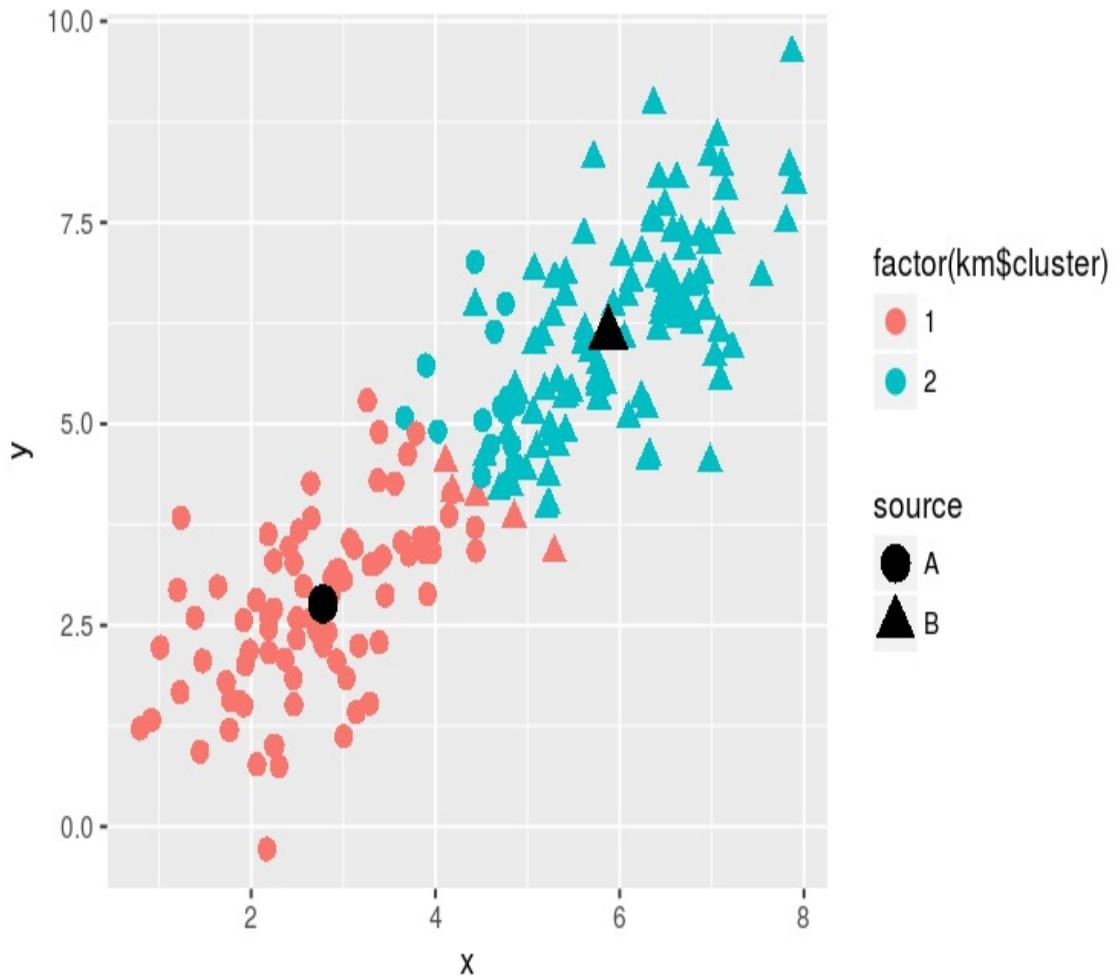
**Figure 10.8** Clustered points. The determined centers of the clusters are shown in black.

The imaginary line separating these two groups is highly dependent on the exact arrangement of the points themselves; each point is determined to be within a group if it is closer to that group than the other groups, but the tipping point for this condition can be extremely delicate. You can also see that some points have been misidentified (some red triangles, some blue circles) if you include the “truth” about which group point belongs to (its source) as a shape aesthetic, as shown in figure [10.9](#):

```
ggplot(spread_data, aes(x, y)) + ①  
  geom_point(size = 3, aes(col = factor(km$cluster), shape  
= source)) +  
  geom_point(data = data.frame(  
    km$centers,  
    z = unique(km$cluster)  
) , aes(x, y, shape = c("A", "B")), size = 5)
```

①

**Don't worry too much about the code used to generate the clusters.**



**Figure 10.9** Notice some misidentified points.

Many classification and clustering algorithms are available, and again, that's a topic worthy of its own book.

### 10.3.3 WORKING WITH MAPS

You've seen how to place data points on a rectangular map and even join them with lines, but that's hardly the only way you might want to visualize data. When working with real-world data, you may come across latitude and longitude coordinates, for example. These shouldn't really go on a flat plot with equal  $x$  and  $y$  scaling; any attempt to do so requires that you consider the *projection* of a globe onto a flat surface.

**DEFINITION** A *projection* is a transformation of the latitudes and longitudes from the surface of a sphere or an ellipsoid into locations on a flat plane. Because the earth is not flat, but paper and screens are, we inevitably require a projection to displace maps, and every projection onto a flat surface creates some distortion.

ggplot2 is able to deal with this through `coord_*` functions. These modify the plot area such that the gridlines follow a certain projection. For example, map data for the states of the USA is accessible from ggplot2 via the `maps` package (once you install it, in addition to ggplot2) and the `map_data()` function:

```
# install.packages("maps") ①
```

①

Note that you don't need to call `library(maps)` — the internals of `map_data()` just require that the package be installed.

You can then plot it with a flat coordinate projection using ggplot2:

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

states <- map_data(map = "state") ①

head(states)
#>      long     lat group order   region subregion
#> 1 -87.46201 30.38968     1     1 alabama      <NA>
#> 2 -87.48493 30.37249     1     2 alabama      <NA>
#> 3 -87.52503 30.37249     1     3 alabama      <NA>
#> 4 -87.53076 30.33239     1     4 alabama      <NA>
#> 5 -87.57087 30.32665     1     5 alabama      <NA>
#> 6 -87.58806 30.32665     1     6 alabama      <NA>
```

①

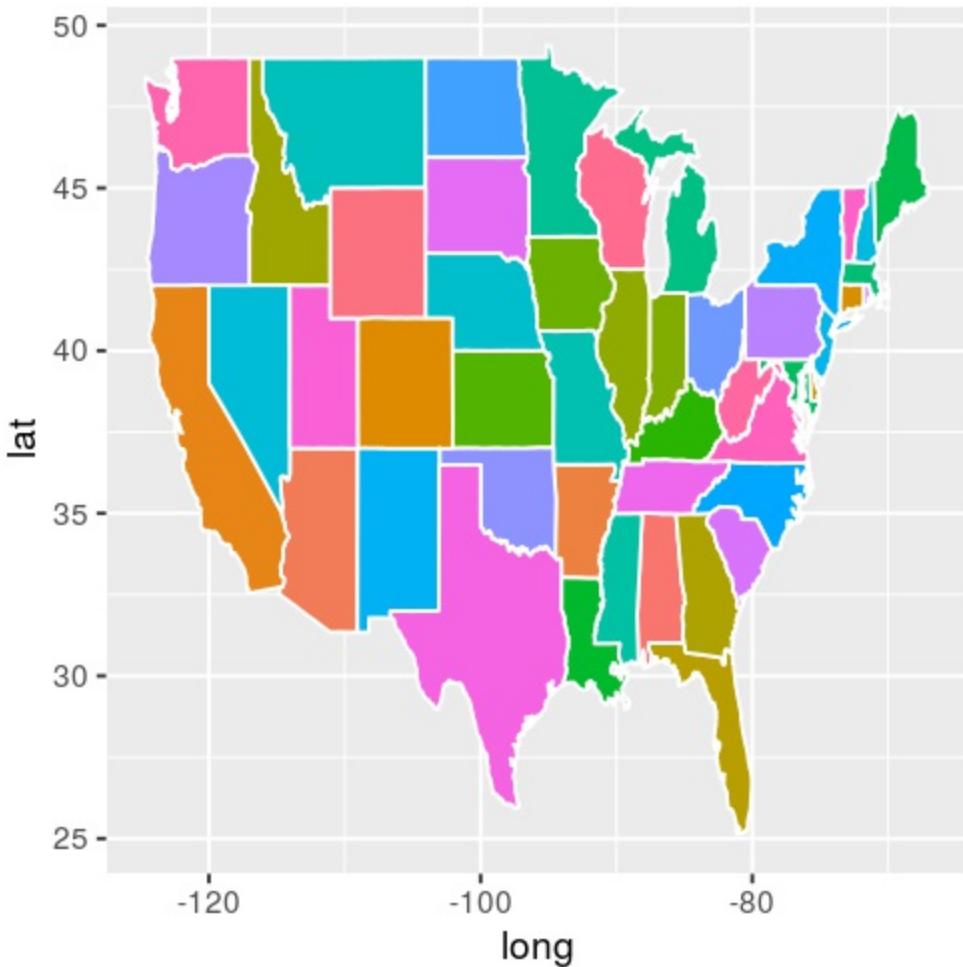
Refer to `?map_data` to see the options available for `map`.

Figure 10.10 shows the result:

```
p <- ggplot(data = states) +  
  geom_polygon(aes(x = long, y = lat,  
                    fill = region, group = group),  
                color = "white") +  
  guides(fill=FALSE) ①  
  
p
```

①

Turns off the fill color legend for now



**Figure 10.10 States of the USA, flat projection**

This looks odd because the aspect ratio of the plot isn't correct. You can enforce a projection on the data that constrains the

aspect ratio (see **figure 10.11**):

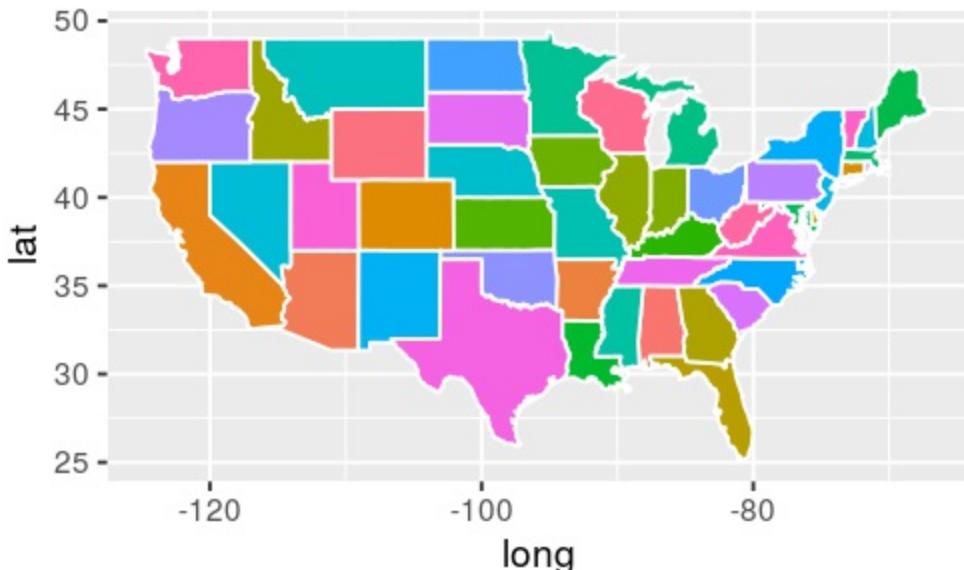
```
p + coord_fixed(1.3)
```

This is roughly correct for this latitude, but to do it correctly you can change to a more suitable projection. To do that, though, the `mapproj` package must also be installed:

```
# install.packages("mapproj")
```

You don't need to use `library()`; `ggplot2` just needs it installed. This allows `ggplot2` to use the `coord_map()` function, resulting in **figure 10.12**:

```
p + coord_map(projection = "albers", lat0 = 30, lat1 = 40)
```



**Figure 10.11** Fixing the aspect ratio begins to help but isn't really accurate.



**Figure 10.12** A proper projection faithfully represents the map.

Several other projections are available. See `?coord_map` for a list and brief descriptions.

You can also use external map providers and the `leaflet` package, producing figure 10.13:

```
# install.packages("leaflet")
library(leaflet)

leaflet() %>%
  addProviderTiles("OpenStreetMap.BlackAndWhite")
%>% ①
  addMarkers(lng = 174.768, lat = -36.852, popup="The
birthplace of R") %>%
  setView(155, -38, zoom = 4)
```

①

Adds OpenStreetMap black and white map tiles. Many providers are available with different map tiles.



**Figure 10.13 A map plot showing the birthplace of R**

R has made great headway toward replacing more traditional geographic information systems (GIS), and working with spatial data is fast becoming easier with new packages such as `sf`.

### 10.3.4 INTERACTING WITH APIs

If you want some information presented to you on a website, you click a link, and your computer/phone does the rest. It communicates with the hosting server, asks for a page, and presents what it receives in a human-readable way. Computers are happy to jump in a step earlier than that and deal with the raw response. When the hosting server is configured just right, it provides a programmatic way to interact with the data, in the form of an API.

In section 7.1.4, you saw how to scrape Wikipedia, but you can get a wider view of the entire site by communicating with its API and searching all of its data rather than starting with a known page. You can request data from servers in general using the `httr` package's `GET` function, which performs the tasks that your browser does when it connects to a server/website.

A simple example of using an API is the Wikipedia `opensearch`

facility, an `httr` call to which looks like this:

```
# install.packages("httr")
library(httr)
wiki_R <- content(
  GET("https://en.wikipedia.org/w/api.php?
action=opensearch&search=R&limit=5"
  )
)
```

This is made up of the API server address (`https://en.wikipedia.org/w/api.php`) and parameters to send as a request. These begin with `?` to identify them — then they're just name/value pairs: `action=opensearch` (use the *opensearch* endpoint), `search=R` (search for pages starting with *R*), and `limit=5` (only return five results). There are other parameters that have default values, and these are usually listed in the API documentation — which it is always a good idea to read. What you get back is a JavaScript Object Notation (JSON) response, part of which contains the content you are after. It also contains details about the connection (whether or not it was successful, authorized, cookies used, HTML headers found, and so on). You can view this content directly in more recent versions of RStudio, as shown in [figure 10.14](#):

```
View(wiki_R)
```

Name	Type	Value
wiki_R	list [4]	List of length 4
[[1]]	character [1]	'R'
[[2]]	list [5]	List of length 5
[[1]]	character [1]	'R'
[[2]]	character [1]	'Russia'
[[3]]	character [1]	'Romania'
[[4]]	character [1]	'Record label'
[[5]]	character [1]	'Rail transport'
[[3]]	list [5]	List of length 5
[[1]]	character [1]	'R (named ar/or ) is the 18th letter of the modern English alphabet and the ISO ...'
[[2]]	character [1]	'Russia (: Russian: Россия, tr. Rossija; IPA: [rə'si婕]), also officially known ...'
[[3]]	character [1]	'Romania ( roh-MAY-nee-ə; Romanian: România [romi'ni.a]) is a sovereign state lo ...'
[[4]]	character [1]	'A record label or record company is a brand or trademark associated with the ma ...'
[[5]]	character [1]	'Rail transport is a means of transferring of passengers and goods on wheeled ve ...'
[[4]]	list [5]	List of length 5
[[1]]	character [1]	'https://en.wikipedia.org/wiki/R'
[[2]]	character [1]	'https://en.wikipedia.org/wiki/Russia'
[[3]]	character [1]	'https://en.wikipedia.org/wiki/Romania'
[[4]]	character [1]	'https://en.wikipedia.org/wiki/Record_label'
[[5]]	character [1]	'https://en.wikipedia.org/wiki/Rail_transport'

**Figure 10.14 Searching the Wikipedia API for pages starting with R**

You can extract elements from this as character vectors:

```
## URL for the letter R page
wiki_R[[4]][[1]]
#> [1] "https://en.wikipedia.org/wiki/R"

## Description for the letter R
strwrap(wiki_R[[3]][[1]], 60) ①
#> [1] "R (named ar/or ) is the 18th letter of the modern
English"
#> [2] "alphabet and the ISO basic Latin alphabet."
```

①

The output is general text, here line-wrapped at 60 characters with `strwrap()`

As another example, the Star Wars API (<https://swapi.co>) provides a wealth of information on the *Star Wars* canon universe. It also allows computers to request this information

directly through API requests. The following gathers the information stored for the second record in the *people* endpoint:

```
C3PO_response <-
content(GET("https://swapi.co/api/people/2/"))
```

And again you can view the resulting content:

```
View(C3PO_response)
```

The result is shown in figure 10.15.

Name	Type	Value
⌚ C3PO_response	list [16]	List of length 16
name	character [1]	'C-3PO'
height	character [1]	'167'
mass	character [1]	'75'
hair_color	character [1]	'n/a'
skin_color	character [1]	'gold'
eye_color	character [1]	'yellow'
birth_year	character [1]	'112BBY'
gender	character [1]	'n/a'
homeworld	character [1]	'https://swapi.co/api/planets/1/'
⌚ films	list [6]	List of length 6
⌚ species	list [1]	List of length 1
vehicles	list [0]	List of length 0
starships	list [0]	List of length 0
created	character [1]	'2014-12-10T15:10:51.357000Z'
edited	character [1]	'2014-12-20T21:17:50.309000Z'
url	character [1]	'https://swapi.co/api/people/2/'

**Figure 10.15** Retrieving the second *people* record from the Star Wars API

Many APIs are available, but it's not uncommon for them to ask that you provide some authorization through a website before using the service.

### 10.3.5 SHARING YOUR PACKAGE

You've made an efficient, tested, working package — super! Are you the only person who will ever want to see what it does or looks like on the inside? Sharing your work is a great way to get involved in the community and get feedback on the approaches you've taken.

There are several ways to share your work. The most common is to host it on a version control system (VCS) hosting site such as GitHub, GitLab, or Bitbucket. This isn't just a convenient way to back up your work — it also allows others (if you choose to allow them) to view your code and perhaps even make suggestions on how you might improve it. If you take this route, it's a great idea to make use of the README functionality; these sites check repositories for a file named README.\* (where the asterisk stands for many different file extensions, often .md, for *Markdown*) and present it as a front page to the repository. Here you can show visitors how to install and use your package, provide notes on how you came to write the code, or maybe cite some relevant sources. This is particularly useful if you're sharing code with colleagues in your organization. You can also set your repository to be private if you have the option, and many organizations host their own internal instances of these services.

If you feel that your package is particularly useful to others, you can request to have it hosted on the Comprehensive R Archive Network (CRAN) so that it can be installed by others using `install.packages()`; but be aware that the maintainers have strict requirements (for very good reasons), and you will need to bring your package up to their standards to be eligible. These standards include passing various checks such as those performed when you use the drop-down menu in the Build pane to access More > Check Package or `devtools::check()`. These extensive checks need to pass without error, warning, or even notes if you expect your code to receive a smooth welcome into the CRAN fold.

Once your code is available for public consumption, it's a good idea to write up a blog post or something to show off what you've made. This not only helps others learn a bit more about your

package, it's also a great way to sharpen your focus on exactly what you want others to think about when they install and use the package. Which functions are they likely to want to use? Is there anything they need to keep in mind? Such thoughts may prompt you to iterate the package and perhaps expand the documentation or provide warnings in the code.

The Twitter R community is generally warmly welcoming and provides some great amplification of interesting work being performed around the globe. If you post about your package, perhaps linking to your hosted repository, make sure to include the `#rstats` hashtag. Blog posts and tweets are frequently added to the R Weekly (<https://rweekly.org>) weekly aggregation through the efforts of editors and community submissions.

## **10.4 More resources**

If you have an idea in mind and want to know what's already available in that space, then the CRAN Task Views page (<https://cran.r-project.org/web/views>) is a good place to start. It offers curated links to packages (with their descriptions) grouped by topic.

Keeping up with new developments in R-related work is made simpler with the help of aggregators like R Weekly (<https://rweekly.org>) and R-bloggers ([www.r-bloggers.com](http://www.r-bloggers.com)). Twitter is a great source of discussion, and there are many dedicated Slack groups hosting more focused discussions.

If you're completely stuck for ideas, then fork/clone someone's R repository and start tinkering. Break it and see if you can fix it. Find something that needs improving. Document something as you learn more about it. rOpenSci (<https://ropensci.org/packages>) hosts many scientifically interesting packages and has a fantastic onboarding process that involves lots of code reviews. Have a look through its resources for inspiration and ideas.

## Terminology

- *Unit test* — An evaluation that confirms whether a particular assumption made by code is true. This is performed at the smallest level possible (a unit of code). Although it may be true that a program as a whole runs, unit testing performed at the function level ensures that a given input produces a given output (or error, warning, and so forth).
- *Profiling* — Automatic examination of the runtimes of individual components within a section of code. Useful for identifying bottlenecks.
- *Regression* — Fitting a model (linear or otherwise) to data, usually with the hope of better understanding a mechanism or producing predictions.
- *Clustering* — Assigning values to one of several groups based on their similarities across various variables.
- *Projection (mapping)* — Presenting a curved surface on a flat plane, involving some transformation.

## Summary

- You can create a minimal R package to contain your code.
- You can document your code reliably with the `roxygen2` package.
- You can perform unit tests and profile some of your code to find bottlenecks.
- You can calculate a simple linear model using regression.
- You can create a projected map visualization.
- You can interact with an API using the `httr` package.

You should now have enough tools at your disposal to go forth and successfully work with the R language. Thank you for following along all this way. I wish you the very best in your endeavors. Please feel free to get in touch with me or tweet (tag

me: @carroll\_jono) if you've found this book useful!

# *appendix A*

## *Installing R*

---

This appendix contains basic information on installing R.

### **Windows**

Visit the R for Windows download page at <https://cran.r-project.org/bin/windows/base>, and download the executable installer. Double-click that file to begin the install process, and follow any prompts.

### **Mac**

Visit the R for Mac OS X/macOS download page at <https://cran.r-project.org/bin/macosx>, and download the relevant binary .pkg file for your system. Install by double-clicking this file and following the prompts.

### **Linux**

For Ubuntu/Debian systems:

```
apt-get update  
apt-get install r-base r-base-dev
```

For Fedora/Red Hat/OpenSUSE systems:

```
yum update  
yum install R
```

### **From source**

As open source software, the *source code* for R can also be downloaded from the SVN repository at <https://svn.r-project.org/R> and compiled. You can even make changes to it as you see fit (if you know how).

R is available as Free Software under the terms of the Free Software Foundation's General Public License in source code form.

*—R-Project, <https://www.r-project.org/about.html>*

Being able to browse the source code can also be handy for advanced debugging, but that is beyond the scope of this book.

# *appendix B*

## *Installing RStudio*

---

Visit the RStudio Downloads page at [www.rstudio.com/products/rstudio/download](http://www.rstudio.com/products/rstudio/download) and follow the instructions to install RStudio on your machine.

### ***Installing RStudio***

- *Windows* — Download the latest .exe file. Double-click that file to begin the install process, and follow any prompts.
- *Mac* — Download the latest .dmg file. Double-click that file to begin the install process, and follow any prompts.
- *Linux* — Download the appropriate file for your machine's architecture (.deb for Ubuntu/Debian, .rpm for Fedora/RedHat/openSUSE) to a folder on your machine (for example, /tmp/).

For Ubuntu/Debian:

```
cd <where_you_saved_the_.deb_file>  
dpkg -i rstudio-v.vv.vvv-aaaaa.deb
```

①

②

**(1)**  
**Saving installation files to /tmp is typical.**

**(2)**  
**v denotes version (for example, 1.1.414), a denotes architecture (i386 or amd64). The exact format of the filename may differ.**

For Fedora/RedHat/openSUSE:

```
cd <where_you_saved_the_.rpm_file>
yum install rstudio-v.vv.vvv-aaaaa.rpm
```

①  
②

①

**Saving installation files to /tmp is typical.**

②

**v denotes version (for example, 1.1.414), a denotes architecture (i686 or x86\_64). The exact format of the filename may differ.**

## Packages used in this book

A wide variety of R packages are both described within and were used to generate the contents of this book. To fully reproduce the output, the following packages must be installed from CRAN:  
tidyverse, here, htmlwidgets, httr, leaflet, mapproj, maps,  
plot3D, reshape2, rex, rio, rlang, roxygen2, testthat, knitr,  
rmarkdown, devtools, and datasauRus.

The specific output produced can vary with the package version, so to ensure that the exact contents of this book are reproducible, a package manifest (relevant to the switchr package) is available from [www.manning.com/books/beyond-spreadsheets-with-r](http://www.manning.com/books/beyond-spreadsheets-with-r) and also on GitHub at <http://mng.bz/5YPz>. The instructions for using it are detailed at <http://mng.bz/6GPy>.

If the switchr package is installed and loaded/attached

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

then the manifest can be used to seed a library:

```
bswrManifest <- readRDS("~/bswrManifest.rds")
①
switchrBaseDir("~/")
②
switchTo("BeyondSpreadsheetsWithR", seed = bswrManifest)
```

③

①

**Loads the manifest file, in this case saved as bswrManifest.rds in the home directory**

②

**Stores the custom library in the home directory**

③

**Creates a new package library and installs all the required packages, with the appropriate versions. Also switches your library to the custom library.**

To stop using the custom library, use this:

```
switchBack()
```

R will once again point to its default library.

## *appendix C*

# *Graphics in base R*

---

This appendix provides an alternative solution to the problem of visualizing data. There are many reasons why `ggplot2` can be seen as a superior method of visualizing data, not least of which is the sophisticated application of the “grammar of graphics” philosophy. That’s not to say that what you’re about to see is useless, but as you learn more about `ggplot2`, you’ll have fewer and fewer reasons to ever use these functions.

The simplest way to generate a base plot is to call `plot()` with vectors of `x` and `y` data, as in [figure C.1](#):

```
plot(x = mtcars$disp, y = mtcars$mpg)
```

You’ll notice a few major differences in how this is written and what is produced. First, since you’re passing vectors (not bare names of columns), you have the dataset explicitly in each argument. The plot shows these argument names as the `x`- and `y`-axis labels (similar to how `ggplot()` did). This isn’t strictly necessary; you could save those to their own variables just the same:

```
disp_data <- mtcars$disp
mpg_data <- mtcars$mpg

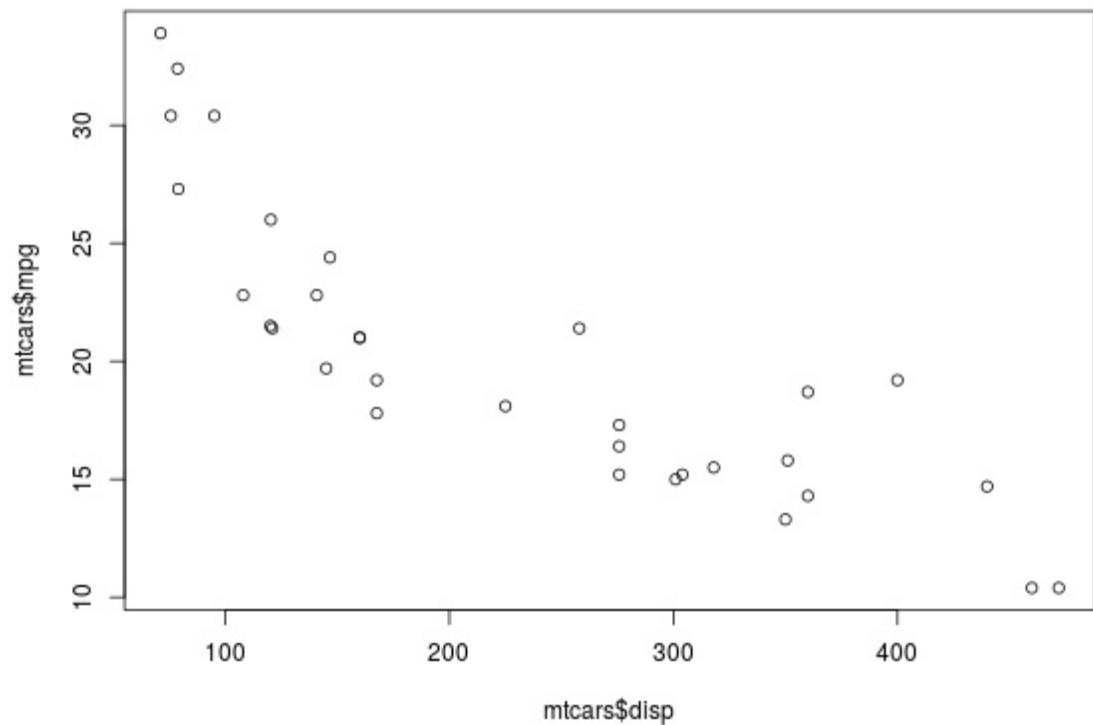
head(mpg_data)

#> [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

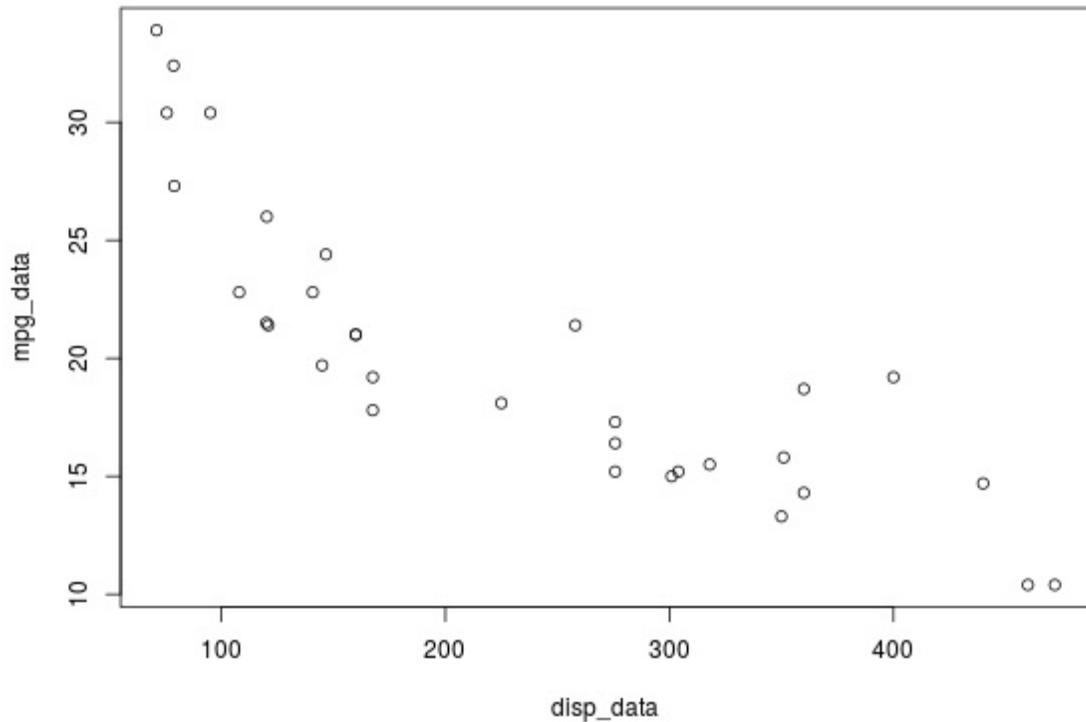
And you could use these objects as the arguments (see [figure C.2](#)):

```
plot(x = disp_data, y = mpg_data)
```

The axis labels now reflect the names of the arguments.



**Figure C.1 A simple base plot**

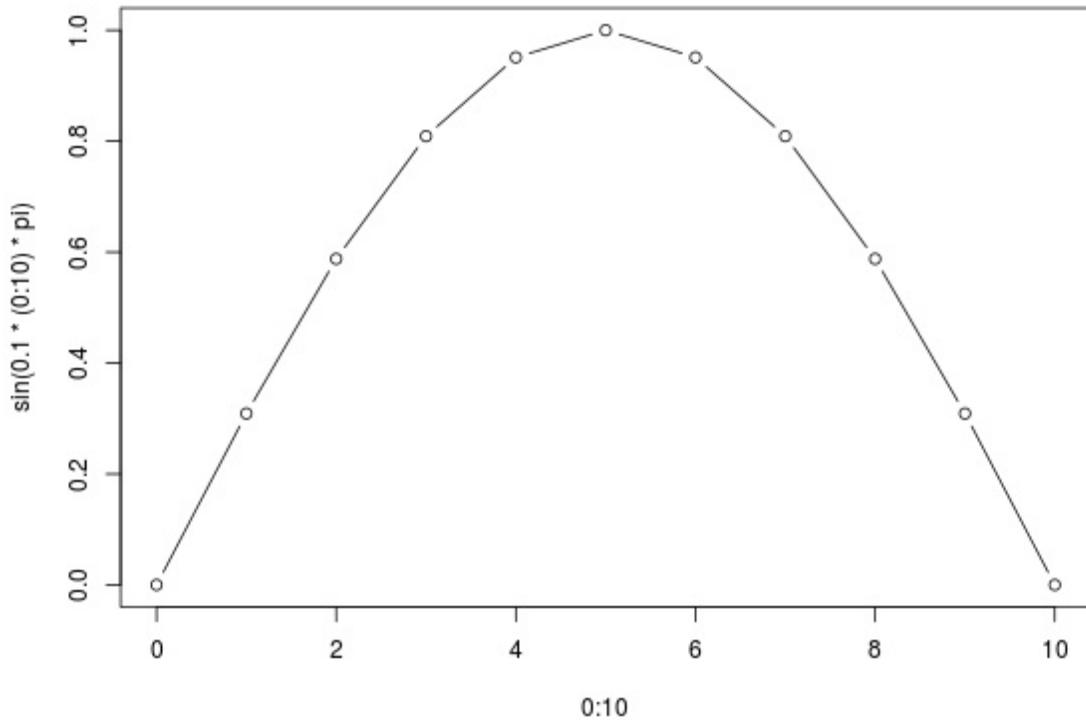


**Figure C.2 Using variables as x and y arguments**

By default, `plot()` plots points (notice I didn't specify either way). For lines joining the points instead, change the `type` argument to "l". To explicitly specify points, set this to "p"; and to include both, set it to "b" (see [figure C.3](#)):

```
plot(x = 0:10, y = sin(0.1*(0:10)*pi), type = "b")
```

Several types of plots are available and are described in `?plot`.



**Figure C.3 Plotting type `b` for both points and lines**

Because the `x` and `y` arguments are disconnected from each other, they don't need to be from the same data source. This shouldn't be appealing; if you're plotting data from multiple sources together, you should probably be joining those sources together beforehand, in which case the tidy aspects come back into play.

It does mean that you need to take care that you keep the pairs of points (each `x` and `y` value) in sync. They also need to have the same number of elements, which is checked by default when you create a `data.frame` or a `tibble`.

You can change the styling of a base plot with arguments to the original call. There are many of these (there's a decent amount of control available), but the most common are the following:

- `col` — Specifies the color of points/lines, as either an integer in `1:8` (selecting a default color) or a character string of a name

from colors()

- pch — Specifies the point character (for points) with the same options as ggplot2
- cex — Specifies the character expansion (magnification, for points)
- lwd — Specifies the line width (for lines)
- lty — Specifies the linetype (for lines)
- xlab — Specifies the x-axis label
- ylab — Specifies the y-axis label
- main — Specifies the plot title

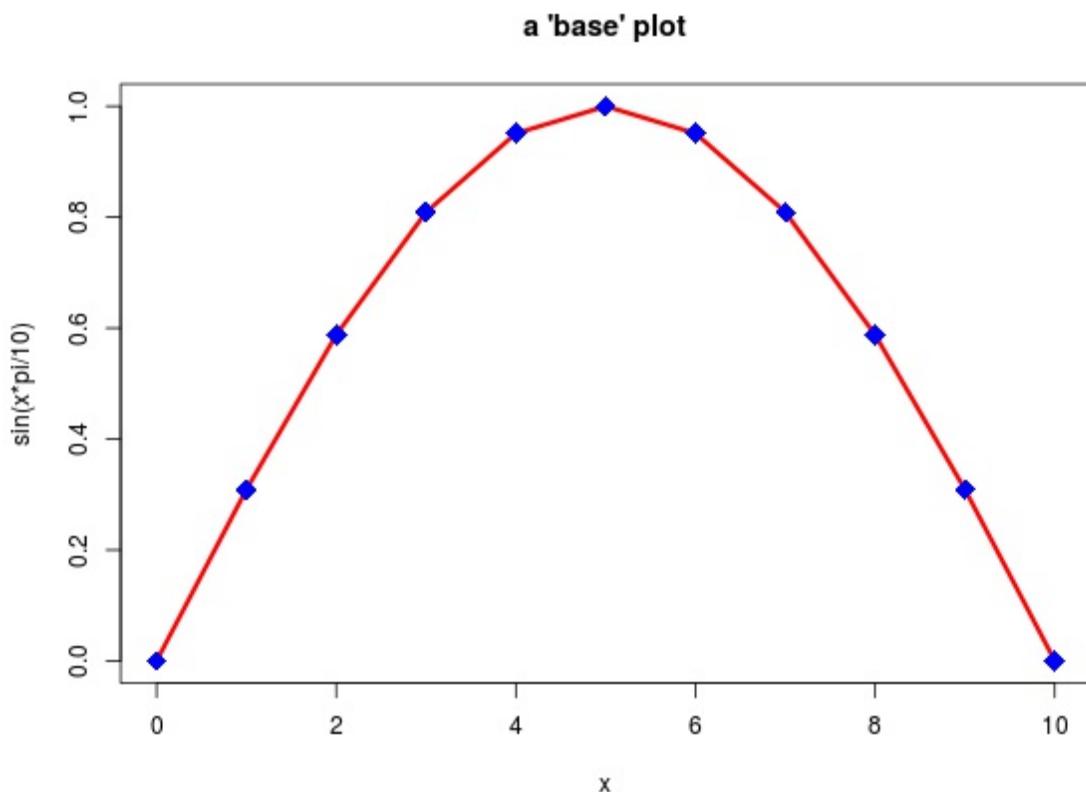
Refer to ?par for a more extensive list of options. These include parameters for modifying the spacing of the plot itself, the layout of multiple plots, and many more options.

To add another feature to a plot once you've generated it, the pen-and-paper aspect of plotting comes up again; instead of using plot(), you can add the feature directly using either points() or lines(). These take the x and y coordinates at which to add the feature, as well as the styling, such as shown in figure C.4:

```
plot(
  x = 0:10,
  y = sin(0.1*(0:10)*pi),
  type = "l",
  col = "red",
  lwd = 3,
  xlab = "x",
  ylab = "sin(x*pi/10)",
  main = "a 'base' plot"
)
points(
  x = 0:10,
  y = sin(0.1*(0:10)*pi),
  pch = 18, # filled diamonds
  cex = 2,
  col = 4    # blue
```

)

The theme of base plots is very minimal, but this too can be altered and configured one parameter at a time.

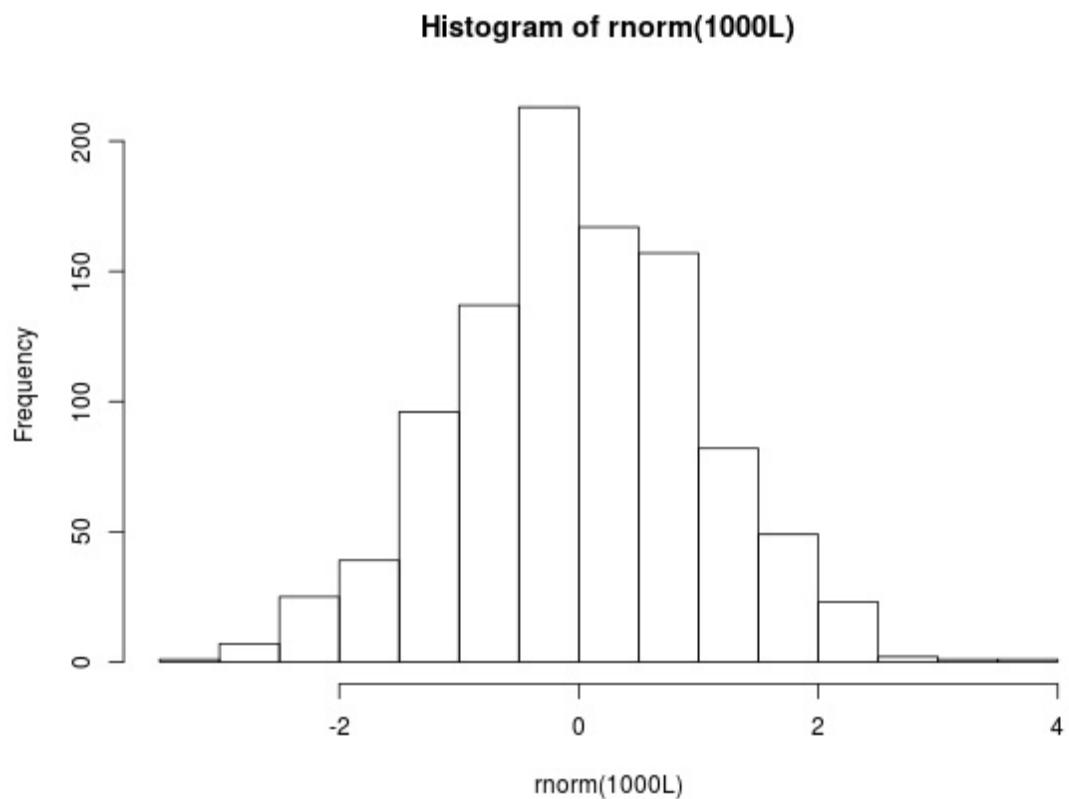


**Figure C.4** Many arguments are available to style the plot.

A significant difference between this method and `ggplot2` is that plots can't be saved as objects; they're drawn to the plot area and then forgotten. You can continue adding features on top of the plot, but to generate any part again, you need to evaluate all the calls that led to it.

There are many other ways to generate plots, such as histograms (see [figure C.5](#)):

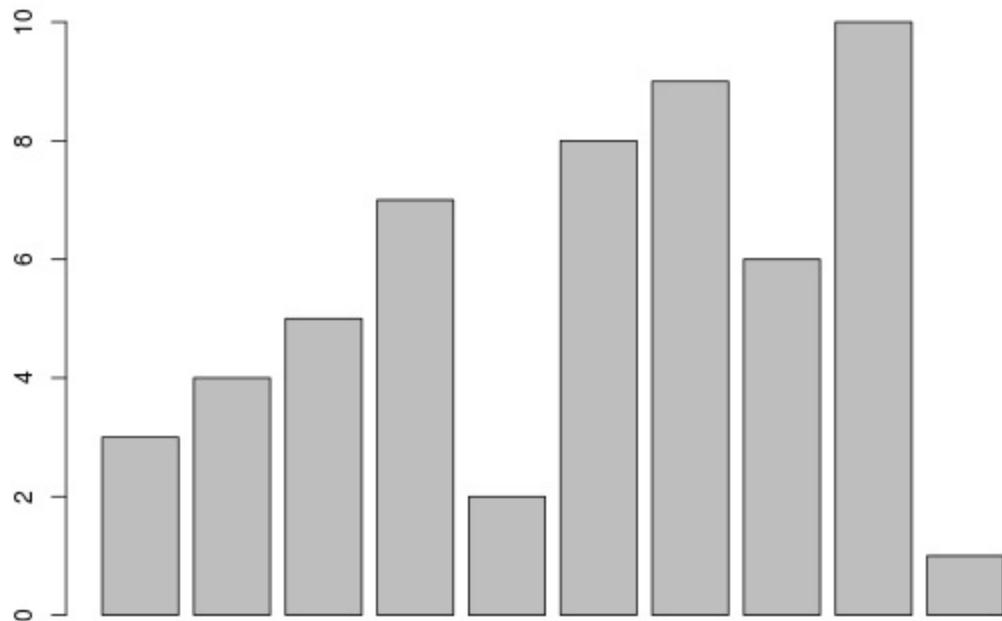
```
set.seed(1)
## histogram of 1000 random, normally-distributed values
hist(rnorm(1000L))
```



**Figure C.5 A base histogram**

Another option is a barplot (see figure C.6):

```
set.seed(1)
barplot(sample(1:10))
```



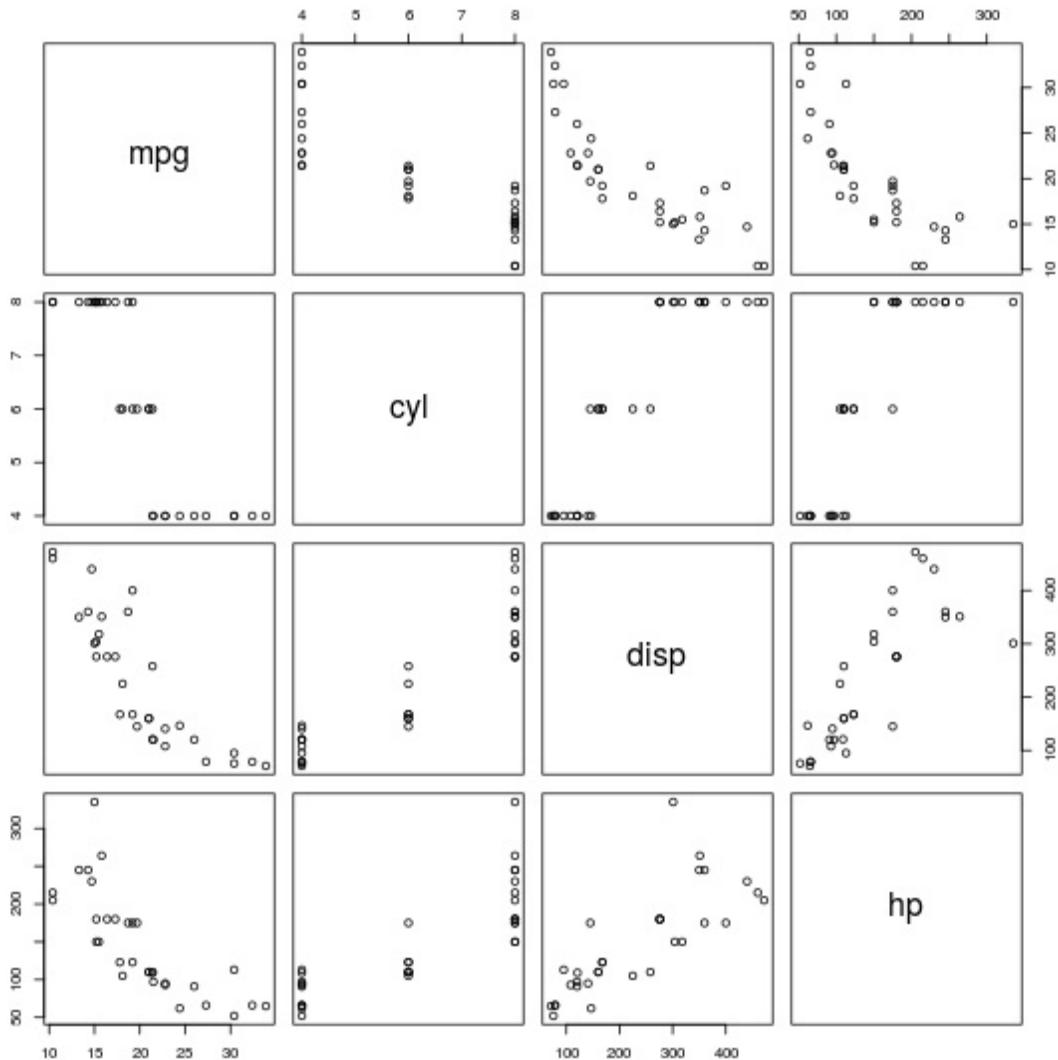
**Figure C.6 A base barplot**

The `plot()` function can apply differently depending on the class of object passed in as the `x` argument. When this is a `data.frame`, a special method is called,<sup>1</sup> which performs some automation of producing a table of scatterplots for inspecting correlation between variables (see figure C.7):

---

<sup>1</sup> This triggers `plot.data.frame()`, though it requires some detective work to see that this is what happens. `methods(plot)` shows a list of the various classes that alter the behavior of `plot()`, including `plot.ggplot`.

```
plot(mtcars[, 1:4])
```



**Figure C.7** Plotting a `data.frame` produces a matrix of variables plotted against each other.

It may be a matter of personal choice whether you feel more comfortable using `plot()` or `ggplot()` to quickly inspect a visualization of data. Personally, I've found learning `ggplot2` to be worth the effort, and I use it to throw together exploratory plots all the time.

When using `base plot`, you need to use a function specific to the file type you want to generate. These are typically named after the filetype, so `png()` generates `plot.png`, `jpeg()` generates `plot.jpeg`, `pdf()` generates `plot.pdf`, and so on. Each of these has

its own options that are specified using arguments, and each takes a filename that determines where the image will be saved. The functions are called *before* saving the image, which changes the output device. This means you won't see the plot when you evaluate the code; it will be written to the file instead. For example:

```
png("mtcars_disp_mpg.png")
plot(x = mtcars$disp, y = mtcars$mpg)
dev.off()
#> null device
#>           1
```

The `null device` message printed means the only remaining active device is device 1 (the null device). You can't shut this one down, and attempting to do so generates an error:

```
dev.off()

#> Error in dev.off() : cannot shut down device 1 (the null
device)
```

Once you've opened a device, you can confirm that by listing the non-null devices with `dev.list()`. If you create two more devices, one of type jpeg and the other of type png:

```
jpeg()
png()
```

you can list them

```
dev.list()
#> jpeg  png
#>     2    3
```

and see that they are numbered sequentially. Close the most recent (the `png` device), leaving just the `jpeg` device active:

```
dev.off()
#> jpeg
#>     2
```

Finally, close the other, once again leaving just the null device:

```
dev.off()
#> null device
#>      1
```

To turn off all graphics devices at once, use `graphics.off()`. You may have created many different devices:

```
tiff()          ①
# <some plot code>
bmp()          ②
# <more plot code>
pdf()          ③
# <more plot code>
png()          ④
# <more plot code>

dev.list()      ⑤
#> tiff   bmp   pdf   png
#>     2     3     4     5
```

①

**Creates a tiff device**

②

**Creates a bmp device**

③

**Creates a pdf device**

④

**Creates a png device**

⑤

**Lists all the active graphics devices**

And you can close all of them at once:

```
graphics.off()
dev.list()
```

#> NULL

# Index

---

## **SYMBOLS**

[[ operator [147](#)  
\$ symbol [151](#)  
+ operator [53, 55](#)  
<- assignment operator [38](#)  
<- assignment operator [44–44](#)  
= assignment operator [44–44](#)

## **A**

abstraction [68, 76](#)  
aes() function [246](#)  
aesthetic mappings [237](#)  
aggregate() function [160, 176](#)  
anti\_join() function [206](#)  
APIs (Application Programming Interfaces) [4, 300–302](#)  
apply() function [219](#)  
areaCircle() function [101, 215](#)  
arguments [44, 73–86, 135–135](#)  
    default [80–82](#)  
    multiple [78–80](#)  
name matching [82–84](#)

partial matching 84–86  
structures as 135–135  
as.character() function 48, 263  
as.integer() function 48  
as.logical() function 49  
assigning (storing) values 38–44  
    assignment operators 44–44  
    unchanging data 43–44  
    variables 38–43  
assignment operators 44–44  
assumptions 4  
asterisks 54  
atomic vectors 180  
attributes 109–110  
attributes() function 128

## B

backslash 52  
backticks 72  
base package 22, 212  
binary files 9  
binary values 26  
braces 226

## C

c() function 33, 106

call stack 100  
case\_when() function 232  
case-sensitive 43  
cat() function 31  
categorical data 31  
categories (factors) data type 31–33  
class attribute 129  
classes 129–135  
    structures as function arguments 135–135  
tibble package 131–135  
closure 152  
clustering 294–297  
coercion (automatic conversion) 62–65, 108–108  
collections 106–112  
    attributes 109–110  
    coercion 108–108  
    missing values 109–109  
    names of vector elements 110–112  
colnames() function 121, 173  
combining data values  
    classes 129–135  
    collections 106–112  
    data.frames 124–129  
    lists 122–124  
    matrices 119–122  
    sequences 112–119

comparisons 59–62  
concatenation 57–58  
conditional evaluation 225–232  
  if conditions 225–229  
  ifelse conditions 229–232  
Console pane 17  
continuous data 31  
control characters 143  
control structures  
  conditional evaluation 225–232  
  looping 213–225  
cos() function 70  
count() function 96  
CRAN (Comprehensive R Archive Network) 12  
create() function 282  
CSV (comma-separated values) 2  
cylinders 254

## D

data 2–10  
  analysis of, benefits 4–6  
  as asset 7–9  
  defined 2–2  
  inspecting 195–196  
  merging 202–207  
  munging 4–4

sources of 2–4  
tidy data principles 182–202  
version control 9–10  
writing from R 207–211

data.frame() function 127

data.frame objects 124–129, 158–169

data.table package 68

datapasta package 189

datasauRus package 308

data types 27–37

- assigning (storing) values 38–44
- categories (factors) 31–33
- dates and times 35–36
- logicals 36–36
- missing values 37–37
- numbers 27–30
- preparing data for plotting 236–236
- specifying 46–50
- strings 31–31

data values

- assigning (storing) 38–44
- coercion 62–65
- combining 106–135
- comparisons 59–62
- mathematical operations 53–55
- operator precedence 56–57

replacing 154–158  
selecting 140–176  
string concatenation 57–58  
summarizing 174–176  
data visualization  
dates and times data type 35–36  
debug mode 101  
debugonce() function 101, 101  
dependency 91  
dev.list() function 315  
devtools package 282  
discrete values 33  
documentation 50  
dot-dot-dot ellipses 83  
dot symbol 41  
double equals character 59  
double numbers 28  
dplyr package 100, 158–158, 167  
dput() function 210  
DSLs (domain-specific languages) 13

## E

Editor pane 17  
encloses 152  
encoding scheme 31  
endsWith() function 140

equal symbol 44  
errors 96–98  
    creating 99–100  
    diagnosing 100–102  
escape character 31  
ESS (Emacs Speaks Statistics) 18  
Excel, R language vs. 177–178  
exponentiate function 103  
exponentiation operator 54  
expressions 16  
extensions  
    analyzing packages 287–291  
    clustering 294–297  
    interacting with APIs 300–302  
    regression 291–294  
    sharing package 302–303  
    working with maps 297–300  
writing packages 282–287

## F

facet\_grid() function 270  
facet\_wrap() function 269  
factor() function 33  
factorial() function 89  
file argument 189, 209  
filter() function 178

first\_letter\_formatted function() 80  
for loops 220–221  
formal arguments 85  
formulas 68  
functions 13, 68–90  
    arguments 73–86  
    built-in 22–23  
    generalizing 103–103  
    inner workings of 70–72  
    R language recognition of 95–95  
    scope 86–90  
    template for 72–73

## G

gather() function 199  
geom\_point() function 241  
ggplot() function 244, 314  
ggplot2 visualization tool 237–275  
    adding bars 251–257  
    adding lines 247–251  
    adding points 241–243  
    facetting 268–272  
    general construction 237–241  
    scales 260–  
    style aesthetics 243–246  
ggsave() function 278

GIS (geographic information systems) 300  
global namespace 95  
global variable 89  
GPL (general purpose language) 13  
graphics.off() function 315  
grep() function 140

## H

help() function 58  
Help and Plot pane 18  
help resources 303–304  
here() function 185  
HTML (Hypertext Markup Language) 191  
htmlwidgets package 308

## I

identical() function 60  
if() construct 227  
if conditions 225–229  
ifelse conditions 229–232  
import() function 190  
impure function 68  
indexing 121–122  
inline documentation 50  
input argument 76  
inspecting data 195–196

install.packages() function 92

integers 28

is.na() function 62

## K

kmeans() function 295

k-means algorithm 295

## L

Last.value variable 40

lazy evaluation 78

lazy loads 96

leaflet package 299

left\_join() function 205

levels() function 33

library() function 92, 284

Linux, installing R language 305–305

list() function 122

lists

general discussion 122–124

selecting data values from 148–152

lm() function 292

load() function 207

local variables 86

logicals data type 36–36

logical types 36

long data [212](#)  
looping [213–221](#)  
    for loops [220–221](#)  
    scope [221–225](#)  
    tidy repetition [215–219](#)  
    vectorization [214–215](#)  
    while loops [224–225](#)

## M

macOS, installing R language [305–305](#)  
magrittr package [167](#)  
make\_a\_list() function [152](#)  
map\_chr() function [216](#)  
map\_df() function [217](#)  
map() function [215](#)  
maps [297–300](#)  
mathematical operations [53–55](#)  
matrices [119–122](#)  
    indexing [121–122](#)  
    naming dimensions [121–122](#)  
    selecting data values from [153–154](#)  
matrix() function [120](#)  
mean() function [84](#)  
merging data [202–207](#)  
message() function [98](#)  
messages [96–98](#)

creating 98–98  
diagnosing 100–102  
metadata 3  
Microsoft Excel, R language vs. 177–178  
Microsoft Windows, installing R language 305–305  
missing values data type 37–37  
    collections 109–109  
    replacing NA 170–171  
mtcars data 44  
mutate() function 162

## N

names of vector elements 110–112  
namespaces 95–96  
NaN (not a number) 37, 231  
narrow scopes 89  
negation operator 171  
nLetters argument 103  
NSE (non-standard evaluation) 106  
NULL value 158  
numbers data type 27–30  
numeric values 28

## O

object argument 209  
operator precedence 56–57

operators 66  
OS X, installing R language 305–305

## P

packages 13, 90–96  
analyzing 287–291  
dplyr package 158–167  
installing 92–94, 308–308  
namespaces 95–96  
profiling 290–291  
purrr package 215–219  
R language recognition of functions 95–95  
sharing 302–303  
tibble package 131–135  
unit testing 288–290  
writing 282–287

panes, RStudio 17

parentheses 54

paste() function 57

pipe operator 164–167

plain-text files 9

plot() function 309

Plots pane 18

plotting

- base R graphics 309–315
- data preparation 235–236

ggplot2 [237–275](#)  
plots as objects [276–278](#)  
saving plots [278–279](#)  
POSIXct type [35](#)  
precedence [66](#)  
print.data() method [130](#)  
print() function [36, 77, 218](#)  
profiling [290–290, 291](#)  
projection [297](#)  
purrr package [215–219](#)

## Q

q() function [20](#)  
quote argument [81](#)

## R

raiseWarning() function [98](#)  
raw data [235](#)  
read\_html() function [192](#)  
read.table() function [188](#)  
readLines() function [190](#)  
real numbers [28](#)  
recursion [89](#)  
recycling [120](#)  
referential transparency [68](#)  
regression [291–294](#)

regular expressions 143–143, 146  
rep() function 221  
replace() function 156  
replacement argument 142  
reproducible research 9–10  
reserved words 42  
resilience 68  
return() function 73, 135  
return value 70  
reusability 68  
RGB (red, green, blue) 245  
right\_join() function 205  
rlang package 162  
R language 11–16  
    cons of 14–14  
    data types 27–37  
    Excel vs. 177–178  
    functions of 14–16  
    ignoring comments and other code 50–51  
    installing 305–306  
    origins of 12–12  
    overview of 13–14  
    pros of 13–14  
    reading data into 187–190  
    recognition of functions 95–95  
    writing from 207–211

rnorm() function [85](#)  
roxygen2 package [308](#)  
RStudio [17–24](#)  
    built-in documentation [23–23](#)  
    built-in packages [22–23](#)  
    installing [307–307](#)  
    vignettes [24–24](#)  
    working within [17–22](#)  
runif() function [82, 85](#)  
rvest package [191](#)  
R Weekly aggregator [303](#)

## S

save() function [207, 209](#)  
saveRDS() function [238](#)  
scale\_color\_continuous() function [266](#)  
scale\_color\_discrete() function [266](#)  
scale function [262](#)  
scales [237](#)  
scope  
    general discussion [86–90](#)  
    looping [221–225](#)  
scraping data [191–194](#)  
scripts [10](#)  
sd (standard deviation) [85](#)  
seed value [82](#)

selecting data values

conditionally 171–174

data.frame objects and dplyr package 158–169

from structures 146–154

replacing NA 170–171

replacing values 154–158

summarizing values 174–176

text processing 140–146

sentinel values 196–198

seq\_along() function 115

seq\_len() function 115

seq() function 138, 250

sequences 112–119

precedence 115

vector functions 116–117

vector math operations 117–119

shortcut function 138

side effects 104

sin() function 71

slash character 52

smallValueCheck() function 101

source() function 16

source code 306

special characters 31

spread() function 199

SQL (Structured Query Language) 202

sqrt() function 215  
startsWith() function 140  
stats package 22, 84  
stop() function 99  
stopifnot() function 99  
str\_view\_all() function 145  
str\_view() function 141  
str() function 29, 38  
streaming data 2  
string concatenation 57–58  
stringr function 145  
stringsAsFactors argument 201  
strongly typed language 27  
strptime function 35  
structures  
    as arguments 135–135  
    conditional evaluation 225–232  
    looping 213–225  
    selecting data values from 146–154  
sub() function 142  
substring() function 102, 103, 142  
substrings 142–142  
sum() function 84, 84  
summarize command 165  
summary() function 174, 195  
suppressMessages() function 100

suppressPackageStartupMessages() function 100  
SVN repository 306  
switchBack() function 308  
switchr package 308  
syntax 11

## T

table() function 195  
tan() function 71  
technical debt 1  
test\_that() function 288  
testing 102–103, 288–290  
text (strings) data type 31–31  
text processing 140–146  
    regular expressions 143–146  
    substrings 142–142  
    text matching 140–141  
    text substitutions 142–143  
theme functions 273  
tibble class 181  
tibble package 131–135  
tidy data principles 182–202  
    converting to tidy data 199–202  
    inspecting data 195–196  
    preparing data for plotting 236–236  
    reading data into R 187–190

scraping data [191–194](#)  
sentinel values [196–198](#)  
stored data formats [186–187](#)  
working directory [183–186](#)  
tidyverse package [308](#)  
tolower() function [80](#)  
toupper() function [80](#)  
traceback() function [100](#)  
truth table [61](#)  
TSV (tab-separated value) [187](#)  
typeof() function [29, 51, 135](#)  
type safety [234](#)

## **U**

unary function [48](#)  
underscores [43](#)  
units [29](#)  
unit test [288](#)  
URL (Uniform Resource Locator) [191](#)

## **V**

variables [13, 38–43](#)  
VCS (version control system) [9, 302](#)  
vectorized operation [119](#)  
vectors [106–108](#)  
    attributes [109–110](#)

functions [116–117](#)  
math operations [117–119](#)  
missing values [109](#)  
names of elements [110–112](#)  
selecting data values from [146–148](#)  
type [108](#)  
version control [9–10](#)  
View() function [196](#)  
vignettes [24–24](#)

## **W**

walk() function [217](#)  
warning() function [98](#)  
warnings [96–98](#)  
    creating [98–98](#)  
    diagnosing [100–102](#)  
weakly typed language [27](#)  
web scraping [191](#)  
which() function [173](#)  
while loops [224–225](#)  
whitespace [48](#)  
wide data [212](#)  
Windows, installing R language [305–305](#)  
working directory [183–186](#)  
Workspace pane [17](#)  
wrap around comments [50](#)

# Lists of Figures, Tables and Listings

---

## ***List of Illustrations***

Figure 1.1: A pattern emerges. These points were generated with a `sin()` function at the values `0, 1, ..., 9`. The smooth `sin()` function is also plotted here.

Figure 1.2: The Broad Street Cholera Map, by John Snow (public domain), via Wikimedia Commons. Dots indicate pump locations, and cases of cholera are marked with stacked bars along streets.

Figure 1.3: Further analyses of the Broad Street cholera data produced using the `HistData` (top) and `cholera` (bottom) R packages

Figure 1.4: Professional users of R ([rstudio.com](http://rstudio.com))

Figure 1.5: Read, evaluate, print, loop

Figure 1.6: R code with some terms identified: variable, assignment operator (`<-`), function, and arguments

Figure 1.7: RStudio panes as they appear in Ubuntu/Linux

Figure 1.8

Figure 1.9

Figure 1.10: Selecting projects from the menu

Figure 1.11: Opening a new script

Figure 1.12: Run button

Figure 2.1: Beatles category index

Figure 2.2: Date types

Figure 2.3: Spreadsheet formula to add two cell values

Figure 2.4: Spreadsheet formula result

Figure 2.5: Components of an assignment operation.

Figure 2.6: APL keyboard, adapted from Wm313 Creative Commons Wikipedia (<http://mng.bz/mzKW>)

Figure 2.7: Environment pane showing the value 4 assigned to the variable x

Figure 2.8: R's process of converting values

Figure 2.9: Adding two values in a spreadsheet

Figure 2.10: Various ways to format numbers in a spreadsheet program

Figure 3.1: Attempting to add a number to a string in a spreadsheet, resulting in an error

Figure 3.3: Ordering of coercion

Figure 4.1

Figure 4.2: Function composition

Figure 4.3

Figure 4.4

Figure 5.1

Figure 5.2: Ordering of coercion

Figure 5.3

Figure 5.4

Figure 5.5

Figure 5.6

Figure 5.7

Figure 6.1

Figure 6.2

Figure 6.3: dplyr verb construction

Figure 6.4

Figure 7.1

Figure 7.2

Figure 7.3

Figure 8.1

Figure 9.1

Figure 9.2

Figure 9.3

Figure 9.4

Figure 9.5

Figure 9.6

Figure 9.7

Figure 9.8

Figure 9.9

Figure 9.10

Figure 9.11

Figure 9.12

Figure 9.13

Figure 9.14

Figure 9.15

Figure 9.16

Figure 9.17

Figure 9.18

Figure 9.19

Figure 9.20

Figure 9.21

Figure 9.22

Figure 9.23

Figure 9.24

Figure 9.25

Figure 9.26

Figure 9.27

Figure 9.28

Figure 9.29

Figure 9.30

Figure 9.31

Figure 9.32

Figure 9.33

Figure 9.34

Figure 9.35

Figure 9.36

Figure 9.37

Figure 9.38

Figure 9.39

Figure 9.40

Figure 9.41

Figure 9.42

Figure 9.43

Figure 9.44

Figure 9.45

Figure 9.46

Figure 9.47

Figure 9.48

Figure 9.49

Figure 9.50

Figure 9.51

Figure 9.52

Figure 9.53

Figure 9.55

Figure 9.56

Figure 9.57

Figure 9.58

Figure 9.59

Figure 9.60

Figure 9.61

Figure 9.62

Figure 10.1

Figure 10.2

Figure 10.3

Figure 10.4

Figure 10.5

Figure 10.6

Figure 10.7

Figure 10.8

Figure 10.9

Figure 10.10

Figure 10.11

Figure 10.12

Figure 10.13

Figure 10.14

Figure 10.15

Figure C.1

Figure C.2

Figure C.3

Figure C.4

Figure C.5

Figure C.6

Figure C.7

## ***List of Tables***

Table 2.1 Data types

Table 2.2 Date and time input format codes

Table 3.1 Truth table for the equals operator (==)

Table 3.2 Truth table for the *and* operator (&)

Table 3.3 Truth table for the *or* operator (|)

Table 6.1 Common regex selectors

Table 6.2 Common regex quantifiers for selecting multiple pattern matches

## ***List of Listings***

Listing 2.1 Try it yourself: assigning to variables

Listing 2.2 Try it yourself: documenting using comments