

O'REILLY®

Second
Edition

R Cookbook

Proven Recipes for Data Analysis,
Statistics & Graphics



J.D. Long &
Paul Teator

R Cookbook

Perform data analysis with R quickly and efficiently with more than 275 practical recipes in this expanded second edition. The R language provides everything you need to do statistical work, but its structure can be difficult to master. These task-oriented recipes make you productive with R immediately. Solutions range from basic tasks to input and output, general statistics, graphics, and linear regression.

Each recipe addresses a specific problem and includes a discussion that explains the solution and provides insight into how it works. If you're a beginner, *R Cookbook* will help get you started. If you're an intermediate user, this book will jog your memory and expand your horizons. You'll get the job done faster and learn more about R in the process.

- Create vectors, handle variables, and perform basic functions
- Simplify data input and output
- Tackle data structures such as matrices, lists, factors, and data frames
- Work with probability, probability distributions, and random variables
- Calculate statistics and confidence intervals and perform statistical tests
- Create a variety of graphic displays
- Build statistical models with linear regressions and analysis of variance (ANOVA)
- Explore advanced statistical techniques, such as finding clusters in your data

DATA / R

US \$69.99 CAN \$92.99

ISBN: 978-1-492-04068-2



5 6 9 9 9

"I learned more from this book than I have from any other programming book I have read."

—David Curran
Cognitive Engineer
at OpenJaw Technologies

J.D. Long works for Renaissance Re in New York City. He's an avid user of Python, R, AWS, and colorful metaphors and is a frequent presenter at R conferences. J.D. is the founder of the Chicago R User Group.

Paul Teator is a quantitative developer with master's degrees in statistics and computer science. Specializing in analytics and software engineering for investment management, he works with hedge funds, market makers, and portfolio managers in the Chicago area.



Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

R Cookbook

*Proven Recipes for Data Analysis,
Statistics, and Graphics*

J.D. Long and Paul Teator

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

R Cookbook

by J.D. Long and Paul Teator

Copyright © 2019 J.D. Long and Paul Teator. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Nicole Tache and Melissa Potter

Production Editor: Kristen Brown

Copyeditor: Rachel Monaghan

Proofreader: Rachel Head

Indexer: Ellen Troutman-Zaig

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

March 2011: First Edition

July 2019: Second Edition

Revision History for the Second Edition

2019-06-21: First Release

2019-11-15: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492040682> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *R Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04068-2

[LSI]

Table of Contents

Welcome to the R Cookbook, 2nd Edition.....	xi
1. Getting Started and Getting Help.....	1
1.1 Downloading and Installing R	2
1.2 Installing RStudio	4
1.3 Starting RStudio	5
1.4 Entering Commands	7
1.5 Exiting from RStudio	9
1.6 Interrupting R	11
1.7 Viewing the Supplied Documentation	12
1.8 Getting Help on a Function	14
1.9 Searching the Supplied Documentation	16
1.10 Getting Help on a Package	17
1.11 Searching the Web for Help	18
1.12 Finding Relevant Functions and Packages	21
1.13 Searching the Mailing Lists	22
1.14 Submitting Questions to Stack Overflow or Elsewhere in the Community	23
2. Some Basics.....	27
2.1 Printing Something to the Screen	27
2.2 Setting Variables	29
2.3 Listing Variables	31
2.4 Deleting Variables	32
2.5 Creating a Vector	34
2.6 Computing Basic Statistics	35
2.7 Creating Sequences	38
2.8 Comparing Vectors	39
2.9 Selecting Vector Elements	41

2.10 Performing Vector Arithmetic	44
2.11 Getting Operator Precedence Right	46
2.12 Typing Less and Accomplishing More	47
2.13 Creating a Pipeline of Function Calls	49
2.14 Avoiding Some Common Mistakes	52
3. Navigating the Software.....	59
3.1 Getting and Setting the Working Directory	59
3.2 Creating a New RStudio Project	60
3.3 Saving Your Workspace	63
3.4 Viewing Your Command History	64
3.5 Saving the Result of the Previous Command	65
3.6 Displaying Loaded Packages via the Search Path	66
3.7 Viewing the List of Installed Packages	68
3.8 Accessing the Functions in a Package	69
3.9 Accessing Built-in Datasets	70
3.10 Installing Packages from CRAN	72
3.11 Installing a Package from GitHub	73
3.12 Setting or Changing a Default CRAN Mirror	74
3.13 Running a Script	76
3.14 Running a Batch Script	77
3.15 Locating the R Home Directory	79
3.16 Customizing R Startup	81
3.17 Using R and RStudio in the Cloud	84
4. Input and Output.....	87
4.1 Entering Data from the Keyboard	87
4.2 Printing Fewer Digits (or More Digits)	88
4.3 Redirecting Output to a File	90
4.4 Listing Files	91
4.5 Dealing with “Cannot Open File” in Windows	94
4.6 Reading Fixed-Width Records	94
4.7 Reading Tabular Data Files	97
4.8 Reading from CSV Files	101
4.9 Writing to CSV Files	103
4.10 Reading Tabular or CSV Data from the Web	104
4.11 Reading Data from Excel	105
4.12 Writing a Data Frame to Excel	107
4.13 Reading Data from a SAS File	109
4.14 Reading Data from HTML Tables	111
4.15 Reading Files with a Complex Structure	113
4.16 Reading from MySQL Databases	118

4.17 Accessing a Database with dbplyr	120
4.18 Saving and Transporting Objects	122
5. Data Structures.....	127
5.1 Appending Data to a Vector	135
5.2 Inserting Data into a Vector	136
5.3 Understanding the Recycling Rule	137
5.4 Creating a Factor (Categorical Variable)	139
5.5 Combining Multiple Vectors into One Vector and a Factor	140
5.6 Creating a List	142
5.7 Selecting List Elements by Position	144
5.8 Selecting List Elements by Name	145
5.9 Building a Name/Value Association List	147
5.10 Removing an Element from a List	149
5.11 Flattening a List into a Vector	150
5.12 Removing NULL Elements from a List	151
5.13 Removing List Elements Using a Condition	152
5.14 Initializing a Matrix	153
5.15 Performing Matrix Operations	155
5.16 Giving Descriptive Names to the Rows and Columns of a Matrix	156
5.17 Selecting One Row or Column from a Matrix	157
5.18 Initializing a Data Frame from Column Data	158
5.19 Initializing a Data Frame from Row Data	160
5.20 Appending Rows to a Data Frame	162
5.21 Selecting Data Frame Columns by Position	165
5.22 Selecting Data Frame Columns by Name	168
5.23 Changing the Names of Data Frame Columns	170
5.24 Removing NAs from a Data Frame	171
5.25 Excluding Columns by Name	172
5.26 Combining Two Data Frames	173
5.27 Merging Data Frames by Common Column	174
5.28 Converting One Atomic Value into Another	177
5.29 Converting One Structured Data Type into Another	178
6. Data Transformations.....	181
6.1 Applying a Function to Each List Element	182
6.2 Applying a Function to Every Row of a Data Frame	184
6.3 Applying a Function to Every Row of a Matrix	185
6.4 Applying a Function to Every Column	186
6.5 Applying a Function to Parallel Vectors or Lists	188
6.6 Applying a Function to Groups of Data	191
6.7 Creating a New Column Based on Some Condition	192

7. Strings and Dates.....	195
7.1 Getting the Length of a String	197
7.2 Concatenating Strings	198
7.3 Extracting Substrings	199
7.4 Splitting a String According to a Delimiter	200
7.5 Replacing Substrings	201
7.6 Generating All Pairwise Combinations of Strings	202
7.7 Getting the Current Date	204
7.8 Converting a String into a Date	204
7.9 Converting a Date into a String	205
7.10 Converting Year, Month, and Day into a Date	206
7.11 Getting the Julian Date	208
7.12 Extracting the Parts of a Date	208
7.13 Creating a Sequence of Dates	210
8. Probability.....	213
8.1 Counting the Number of Combinations	215
8.2 Generating Combinations	216
8.3 Generating Random Numbers	217
8.4 Generating Reproducible Random Numbers	219
8.5 Generating a Random Sample	220
8.6 Generating Random Sequences	221
8.7 Randomly Permuting a Vector	222
8.8 Calculating Probabilities for Discrete Distributions	223
8.9 Calculating Probabilities for Continuous Distributions	225
8.10 Converting Probabilities to Quantiles	227
8.11 Plotting a Density Function	228
9. General Statistics.....	233
9.1 Summarizing Your Data	235
9.2 Calculating Relative Frequencies	237
9.3 Tabulating Factors and Creating Contingency Tables	238
9.4 Testing Categorical Variables for Independence	239
9.5 Calculating Quantiles (and Quartiles) of a Dataset	240
9.6 Inverting a Quantile	241
9.7 Converting Data to z-Scores	242
9.8 Testing the Mean of a Sample (t-Test)	243
9.9 Forming a Confidence Interval for a Mean	244
9.10 Forming a Confidence Interval for a Median	246
9.11 Testing a Sample Proportion	247
9.12 Forming a Confidence Interval for a Proportion	248
9.13 Testing for Normality	249

9.14 Testing for Runs	250
9.15 Comparing the Means of Two Samples	252
9.16 Comparing the Locations of Two Samples Nonparametrically	254
9.17 Testing a Correlation for Significance	255
9.18 Testing Groups for Equal Proportions	257
9.19 Performing Pairwise Comparisons Between Group Means	258
9.20 Testing Two Samples for the Same Distribution	260
10. Graphics.....	263
10.1 Creating a Scatter Plot	267
10.2 Adding a Title and Labels	268
10.3 Adding (or Removing) a Grid	270
10.4 Applying a Theme to a ggplot Figure	274
10.5 Creating a Scatter Plot of Multiple Groups	278
10.6 Adding (or Removing) a Legend	280
10.7 Plotting the Regression Line of a Scatter Plot	284
10.8 Plotting All Variables Against All Other Variables	288
10.9 Creating One Scatter Plot for Each Group	290
10.10 Creating a Bar Chart	292
10.11 Adding Confidence Intervals to a Bar Chart	295
10.12 Coloring a Bar Chart	298
10.13 Plotting a Line from x and y Points	300
10.14 Changing the Type, Width, or Color of a Line	302
10.15 Plotting Multiple Datasets	305
10.16 Adding Vertical or Horizontal Lines	307
10.17 Creating a Boxplot	309
10.18 Creating One Boxplot for Each Factor Level	311
10.19 Creating a Histogram	313
10.20 Adding a Density Estimate to a Histogram	315
10.21 Creating a Normal Quantile–Quantile Plot	317
10.22 Creating Other Quantile–Quantile Plots	319
10.23 Plotting a Variable in Multiple Colors	322
10.24 Graphing a Function	325
10.25 Displaying Several Figures on One Page	328
10.26 Writing Your Plot to a File	331
11. Linear Regression and ANOVA.....	333
11.1 Performing Simple Linear Regression	335
11.2 Performing Multiple Linear Regression	337
11.3 Getting Regression Statistics	339
11.4 Understanding the Regression Summary	342
11.5 Performing Linear Regression Without an Intercept	345

11.6 Regressing Only Variables That Highly Correlate with Your Dependent Variable	347
11.7 Performing Linear Regression with Interaction Terms	350
11.8 Selecting the Best Regression Variables	352
11.9 Regressing on a Subset of Your Data	357
11.10 Using an Expression Inside a Regression Formula	358
11.11 Regressing on a Polynomial	360
11.12 Regressing on Transformed Data	361
11.13 Finding the Best Power Transformation (Box–Cox Procedure)	363
11.14 Forming Confidence Intervals for Regression Coefficients	368
11.15 Plotting Regression Residuals	369
11.16 Diagnosing a Linear Regression	371
11.17 Identifying Influential Observations	375
11.18 Testing Residuals for Autocorrelation (Durbin–Watson Test)	376
11.19 Predicting New Values	378
11.20 Forming Prediction Intervals	379
11.21 Performing One-Way ANOVA	380
11.22 Creating an Interaction Plot	382
11.23 Finding Differences Between Means of Groups	383
11.24 Performing Robust ANOVA (Kruskal–Wallis Test)	386
11.25 Comparing Models by Using ANOVA	388
12. Useful Tricks.....	391
12.1 Peeking at Your Data	391
12.2 Printing the Result of an Assignment	394
12.3 Summing Rows and Columns	394
12.4 Printing Data in Columns	395
12.5 Binning Your Data	396
12.6 Finding the Position of a Particular Value	397
12.7 Selecting Every nth Element of a Vector	398
12.8 Finding Minimums or Maximums	399
12.9 Generating All Combinations of Several Variables	401
12.10 Flattening a Data Frame	402
12.11 Sorting a Data Frame	403
12.12 Stripping Attributes from a Variable	404
12.13 Revealing the Structure of an Object	405
12.14 Timing Your Code	408
12.15 Suppressing Warnings and Error Messages	410
12.16 Taking Function Arguments from a List	411
12.17 Defining Your Own Binary Operators	413
12.18 Suppressing the Startup Message	414
12.19 Getting and Setting Environment Variables	415

12.20 Use Code Sections	416
12.21 Executing R in Parallel Locally	417
12.22 Executing R in Parallel Remotely	420
13. Beyond Basic Numerics and Statistics.....	425
13.1 Minimizing or Maximizing a Single-Parameter Function	425
13.2 Minimizing or Maximizing a Multiparameter Function	426
13.3 Calculating Eigenvalues and Eigenvectors	428
13.4 Performing Principal Component Analysis	429
13.5 Performing Simple Orthogonal Regression	430
13.6 Finding Clusters in Your Data	433
13.7 Predicting a Binary-Valued Variable (Logistic Regression)	436
13.8 Bootstrapping a Statistic	438
13.9 Factor Analysis	441
14. Time Series Analysis.....	447
14.1 Representing Time Series Data	449
14.2 Plotting Time Series Data	452
14.3 Extracting the Oldest or Newest Observations	454
14.4 Subsetting a Time Series	456
14.5 Merging Several Time Series	458
14.6 Filling or Padding a Time Series	460
14.7 Lagging a Time Series	463
14.8 Computing Successive Differences	464
14.9 Performing Calculations on Time Series	466
14.10 Computing a Moving Average	467
14.11 Applying a Function by Calendar Period	469
14.12 Applying a Rolling Function	471
14.13 Plotting the Autocorrelation Function	473
14.14 Testing a Time Series for Autocorrelation	475
14.15 Plotting the Partial Autocorrelation Function	476
14.16 Finding Lagged Correlations Between Two Time Series	478
14.17 Detrending a Time Series	479
14.18 Fitting an ARIMA Model	482
14.19 Removing Insignificant ARIMA Coefficients	486
14.20 Running Diagnostics on an ARIMA Model	487
14.21 Making Forecasts from an ARIMA Model	490
14.22 Plotting a Forecast	491
14.23 Testing for Mean Reversion	492
14.24 Smoothing a Time Series	495

15. Simple Programming.....	499
15.1 Choosing Between Two Alternatives: if/else	500
15.2 Iterating with a Loop	502
15.3 Defining a Function	503
15.4 Creating a Local Variable	505
15.5 Choosing Between Multiple Alternatives: switch	506
15.6 Defining Defaults for Function Parameters	507
15.7 Signaling Errors	508
15.8 Protecting Against Errors	509
15.9 Creating an Anonymous Function	510
15.10 Creating a Collection of Reusable Functions	511
15.11 Automatically Reindenting Code	513
16. R Markdown and Publishing.....	515
16.1 Creating a New Document	516
16.2 Adding a Title, Author, or Date	518
16.3 Formatting Document Text	520
16.4 Inserting Document Headings	521
16.5 Inserting a List	521
16.6 Showing Output from R Code	523
16.7 Controlling Which Code and Results Are Shown	525
16.8 Inserting a Plot	526
16.9 Inserting a Table	530
16.10 Inserting a Table of Data	531
16.11 Inserting Math Equations	534
16.12 Generating HTML Output	535
16.13 Generating PDF Output	536
16.14 Generating Microsoft Word Output	539
16.15 Generating Presentation Output	546
16.16 Creating a Parameterized Report	548
16.17 Organizing Your R Markdown Workflow	552
Index.....	555

Welcome to the R Cookbook, 2nd Edition

R is a powerful tool for statistics, graphics, and statistical programming. It is used by tens of thousands of people daily to perform serious statistical analyses. It is a free, open source system whose implementation is the collective accomplishment of many intelligent, hard-working people. There are more than 10,000 available add-on packages, and R is a serious rival to all commercial statistical packages.

But R can be frustrating. It's not obvious how to accomplish many tasks, even simple ones. The simple tasks are easy once you know how, yet figuring out that "how" can be maddening.

This book is full of how-to recipes, each of which solves a specific problem. Each recipe includes a quick introduction to the solution followed by a discussion that aims to unpack the solution and give you some insight into how it works. We know these recipes are useful and we know they work, because we use them ourselves.

The range of recipes is broad. It starts with basic tasks before moving on to input and output, general statistics, graphics, and linear regression. Any significant work with R will involve most or all of these areas.

If you are a beginner, then this book will get you started faster. If you are an intermediate user, this book will be useful for expanding your horizons and jogging your memory ("How do I do that Kolmogorov–Smirnov test again?").

The book is not a tutorial on R, although you will learn something by studying the recipes. It is not a reference manual, but it does contain a lot of useful information. It is not a book on programming in R, although many recipes are useful inside R scripts.

Finally, this book is not an introduction to statistics. Many recipes assume that you are familiar with the underlying statistical procedure, if any, and just want to know how it's done in R.

The Recipes

Most recipes use one or two R functions to solve a specific problem. It's important to remember that we do not describe the functions in detail; rather, we describe just enough to solve the immediate problem. Nearly every such function has additional capabilities beyond those described here, and some have amazing capabilities. We strongly urge you to read the functions' help pages. You will likely learn something valuable.

Each recipe presents one way to solve a particular problem. Of course, there are likely several reasonable solutions to each problem. When we knew of multiple solutions, we generally selected the simplest one. For any given task, you can probably discover several alternative solutions yourself. This is a cookbook, not a bible.

In particular, R has literally thousands of downloadable add-on packages, many of which implement alternative algorithms and statistical methods. This book concentrates on the core functionality available through the basic distribution combined with several important packages known collectively as the *tidyverse*.

The most concise definition of the tidyverse comes from [Hadley Wickham](#), its originator and one of its core maintainers:

The tidyverse is a set of packages that work in harmony because they share common data representations and API design. The `tidyverse` package is designed to make it easy to install and load core packages from the tidyverse in a single command. The best place to learn about all the packages in the tidyverse and how they fit together is [R for Data Science](#).

A Note on Terminology

The goal of every recipe is to solve a problem and solve it quickly. Rather than labouring in tedious prose, we occasionally streamline the description with terminology that is correct but not precise. A good example is the term *generic function*. We refer to `print(x)` and `plot(x)` as generic functions because they work for many kinds of `x`, handling each kind appropriately. A computer scientist would wince at our terminology because, strictly speaking, these are not simply “functions”; they are polymorphic methods with dynamic dispatching. But if we carefully unpacked every such technical detail, the essential solutions would be buried in the technicalities. So we just call them functions, which we think is more readable.

Another example, taken from statistics, is the complexity surrounding the semantics of statistical hypothesis testing. Using the strict language of probability theory would obscure the practical application of some tests, so we use more colloquial language when describing each statistical test. See the introduction to [Chapter 9](#) for more about how hypothesis tests are presented in the recipes.

Our goal is to make the power of R available to a wide audience by writing readably, not formally. We hope that experts in their respective fields will understand if our terminology is occasionally informal.

Software and Platform Notes

The base distribution of R has frequent and planned releases, but the language definition and core implementation are stable. The recipes in this book should work with any recent release of the base distribution.

Some recipes have platform-specific considerations, and we have carefully noted them. Those recipes mostly deal with software issues, such as installation and configuration. As far as we know, all other recipes will work on all three major platforms for R: Windows, macOS, and Linux/Unix.

Other Resources

Here are a few suggestions for further reading, if you'd like to dig a little deeper:

On the web

The mother ship for all things R is the [R project site](#). From there you can download R for your platform, add-on packages, documentation, and source code as well as many other resources.

Beyond the R project site, we recommend using an R-specific search engine, such as [RSeek](#), created by Sasha Goodman. You can use a generic search engine, such as Google, but the “R” search term brings up too much extraneous stuff. See [Recipe 1.11](#) for more about searching the web.

Reading blogs is a great way to learn about R and stay abreast of leading-edge developments. There are surprisingly many such blogs, so we recommend following two blogs-of-blogs: [R-bloggers](#), created by Tal Galili, and [PlanetR](#). By subscribing to their RSS feeds, you will be notified of interesting and useful articles from dozens of websites.

R books

There are many, many books about learning and using R. Listed here are a few that we have found useful. Note that the R project site contains an [extensive bibliography of books related to R](#).

[R for Data Science](#), by Hadley Wickham and Garrett Grolemund (O'Reilly), is an excellent introduction to the tidyverse packages, especially for using them in data analysis and statistics. It is also available [online](#).

We find the [R Graphics Cookbook, 2nd ed.](#), by Winston Chang (O'Reilly), indispensable for creating graphics. The book [ggplot2: Elegant Graphics for Data](#)

Analysis by Hadley Wickham (Springer) is the definitive reference for the graphics package `ggplot2`, which we use in this book.

Anyone doing serious graphics work in R will want *R Graphics* by Paul Murrell (Chapman & Hall/CRC).

R in a Nutshell, by Joseph Adler (O'Reilly), is the quick tutorial and reference you'll keep by your side. It covers many more topics than this cookbook.

New books on programming in R appear regularly. We suggest *Hands On Programming with R* by Garrett Grolemund (O'Reilly) for an introduction, or *The Art of R Programming* by Normal Matloff (No Starch Press). Hadley Wickham's *Advanced R* (Chapman & Hall/CRC) is available either as a printed book or [free online](#) and is a great deeper dive into advanced R topics. *Efficient R Programming*, by Colin Gillespie and Robin Lovelace (O'Reilly), is another good guide to learning the deeper concepts about R programming.

Modern Applied Statistics with S, 4th ed., by William Venables and Brian Ripley (Springer), uses R to illustrate many advanced statistical techniques. The book's functions and datasets are available in the `MASS` package, which is included in the standard distribution of R.

Serious geeks can download the [R Language Definition](#) from the R Core Team. The Definition is a work in progress, but it can answer many of your detailed questions regarding R as a programming language.

Statistics books

For learning statistics, a great choice is *Using R for Introductory Statistics* by John Verzani (Chapman & Hall/CRC). It teaches statistics and R together, giving you the necessary computer skills to apply the statistical methods.

You will need a good statistics textbook or reference book to accurately interpret the statistical tests performed in R. There are many such fine books—far too many for us to recommend any one above the others.

Increasingly, statistics authors are using R to illustrate their methods. If you work in a specialized field, then you will likely find a useful and relevant book in the [R project bibliography](#).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, packages, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, source code for the book, exercises, etc.) is available for download at <http://rc2e.com>. The Twitter account for content associated with this book is [@R_cookbook](#).

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*R Cookbook*, 2nd ed., by J.D. Long and Paul Teator. Copyright 2019 J.D. Long and Paul Teator, 978-1-492-04068-2.”

If you feel your use of code examples falls outside fair use or the permission just described, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/RCookbook_2e.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

With gratitude we thank the R community in general and the R Core Team in particular. Their selfless contributions are enormous. The world of statistics is benefiting tremendously from their work. The R Studio Community Discussion participants were very helpful in workshopping ideas around how to explain many things. And the staff and leadership of R Studio were supportive in so many little and big ways. We owe them a debt of gratitude for all they have given back to the R community.

We wish to thank the book's technical reviewers: David Curran, Justin Shea, and MAJ Dusty Turner. Their feedback was critical for improving the quality, accuracy, and usefulness of this book. Our editors, Melissa Potter and Rachel Monaghan, were helpful beyond imagination and they frequently prevented us from publicly demonstrating our ignorance. Our production editor, Kristen Brown, is the envy of all technical authors because of her speed and her proficiency with both Markdown and Git.

Paul would like to thank his family for their support and patience during the creation of this book.

J.D. would like to thank his wife Mary Beth and daughter Ada for their patience with all the early mornings and weekends that he spent with his face in the laptop working on this book.

Getting Started and Getting Help

This chapter sets the groundwork for the other chapters. It explains how to download, install, and run R.

More importantly, it also explains how to get answers to your questions. The R community provides a wealth of documentation and assistance. You are not alone. Here are some common sources of help:

Local, installed documentation

When you install R on your computer, a mass of documentation is also installed. You can browse the local documentation ([Recipe 1.7](#)) and search it ([Recipe 1.9](#)). We are amazed how often we search the web for an answer only to discover it was already available in the installed documentation.

Task views

A [task view](#) describes packages that are specific to one area of statistical work, such as econometrics, medical imaging, psychometrics, or spatial statistics. Each task view is written and maintained by an expert in the field. There are more than 35 such task views, so there is likely to be one or more for your areas of interest. We recommend that every beginner find and read at least one task view in order to gain a sense of R's possibilities ([Recipe 1.12](#)).

Package documentation

Most packages include useful documentation. Many also include overviews and tutorials, called *vignettes* in the R community. The documentation is kept with the packages in package repositories such as [CRAN](#), and it is automatically installed on your machine when you install a package.

Question and answer (Q&A) websites

On a Q&A site, anyone can post a question, and knowledgeable people can respond. Readers vote on the answers, so the best answers tend to emerge over time. All this information is tagged and archived for searching. These sites are a cross between a mailing list and a social network; **Stack Overflow** is the canonical example.

The web

The web is loaded with information about R, and there are R-specific tools for searching it ([Recipe 1.11](#)). The web is a moving target, so be on the lookout for new, improved ways to organize and search information regarding R.

Mailing lists

Volunteers have generously donated many hours of time to answer beginners' questions that are posted to the R mailing lists. The lists are archived, so you can search the archives for answers to your questions ([Recipe 1.13](#)).

1.1 Downloading and Installing R

Problem

You want to install R on your computer.

Solution

Windows and macOS users can download R from CRAN, the Comprehensive R Archive Network. Linux and Unix users can install R packages using their package management tool.

Windows

1. Open <http://www.r-project.org/> in your browser.
2. Click on “CRAN.” You’ll see a list of mirror sites, organized by country.
3. Select a site near you or the top one listed as “0-Cloud,” which tends to work well for most locations (<https://cloud.r-project.org/>).
4. Click on “Download R for Windows” under “Download and Install R.”
5. Click on “base.”
6. Click on the link for downloading the latest version of R (an .exe file).
7. When the download completes, double-click on the .exe file and answer the usual questions.

macOS

1. Open <http://www.r-project.org/> in your browser.
2. Click on “CRAN.” You’ll see a list of mirror sites, organized by country.
3. Select a site near you or the top one listed as “0-Cloud,” which tends to work well for most locations.
4. Click on “Download R for (Mac) OS X.”
5. Click on the *.pkg* file for the latest version of R, under “Latest release;” to download it.
6. When the download completes, double-click on the *.pkg* file and answer the usual questions.

Linux or Unix

The major Linux distributions have packages for installing R. **Table 1-1** shows some examples.

Table 1-1. Linux distributions

Distribution	Package name
Ubuntu or Debian	r-base
Red Hat or Fedora	R.i386
SUSE	R-base

Use the system’s package manager to download and install the package. Normally, you will need the root password or `sudo` privileges; otherwise, ask a system administrator to perform the installation.

Discussion

Installing R on Windows or macOS is straightforward because there are prebuilt binaries (compiled programs) for those platforms. You need only follow the preceding instructions. The CRAN web pages also contain links to installation-related resources, such as frequently asked questions (FAQs) and tips for special situations (“Does R run under Windows Vista/7/8/Server 2008?”), that you may find useful.

The best way to install R on Linux or Unix is by using your Linux distribution package manager to install R as a package. The distribution packages greatly streamline both the initial installation and subsequent updates.

On Ubuntu or Debian, use `apt-get` to download and install R. Run under `sudo` to have the necessary privileges:

```
$ sudo apt-get install r-base
```

On Red Hat or Fedora, use yum:

```
$ sudo yum install R.i386
```

Most Linux platforms also have graphical package managers, which you might find more convenient.

Beyond the base packages, we recommend installing the documentation packages, too. We like to install `r-base-html` (because we like browsing the hyperlinked documentation) as well as `r-doc-html`, which installs the important R manuals locally:

```
$ sudo apt-get install r-base-html r-doc-html
```

Some Linux repositories also include prebuilt copies of R packages available on CRAN. We don't use them because we'd rather get software directly from CRAN itself, which usually has the freshest versions.

In rare cases, you may need to build R from scratch. You might have an obscure, unsupported version of Unix, or you might have special considerations regarding performance or configuration. The build procedure on Linux or Unix is quite standard. Download the tarball from the home page of your CRAN mirror; it'll be called something like `R-3.5.1.tar.gz`, except the 3.5.1 will be replaced by the latest version. Unpack the tarball, look for a file called `INSTALL`, and follow the directions.

See Also

R in a Nutshell by Joseph Adler (O'Reilly) contains more details on downloading and installing R, including instructions for building the Windows and macOS versions. Perhaps the ultimate guide is the one entitled "[R Installation and Administration](#)", available on CRAN, which describes building and installing R on a variety of platforms.

This recipe is about installing the base package. See [Recipe 3.10](#) for installing add-on packages from CRAN.

1.2 Installing RStudio

Problem

You want a more comprehensive integrated development environment (IDE) than the R default. In other words, you want to install RStudio Desktop.

Solution

Over the past few years RStudio has become the most widely used IDE for R. We are of the opinion that almost all R work should be done in the RStudio Desktop IDE,

unless there is a compelling reason to do otherwise. RStudio makes multiple products, including RStudio Desktop, RStudio Server, and RStudio Shiny Server, just to name a few. For this book we will use the term *RStudio* to mean RStudio Desktop, though most concepts apply to RStudio Server as well.

To install RStudio, download the latest installer for your platform from the [RStudio website](#).

The RStudio Desktop Open Source License version is free to download and use.

Discussion

This book was written and built using RStudio version 1.2.x and R versions 3.5.x. New versions of RStudio are released every few months, so be sure to update regularly. Note that RStudio works with whichever version of R you have installed, so updating to the latest version of RStudio does *not* upgrade your version of R. R must be upgraded separately.

Interacting with R is slightly different in RStudio than in the built-in R user interface. For this book, we've elected to use RStudio for all examples.

1.3 Starting RStudio

Problem

You want to run RStudio on your computer.

Solution

A common mistake made by new users of R and RStudio is to accidentally start R when they intended to start RStudio. The easiest way to ensure you're actually starting RStudio is to search for RStudio on your desktop, then use whatever method your OS provides for pinning the icon somewhere easy to find later:

Windows

Click on the Start Screen menu in the lower-left corner of the screen. In the search box, type **RStudio**.

macOS

Look in your launchpad for the RStudio app or press Cmd-space (Cmd is the command or ⌘ key) and type **RStudio** to search using Spotlight Search.

Ubuntu

Press Alt-F1 and type **RStudio** to search for RStudio.

Discussion

It's easy to get confused between R and RStudio because, as you can see in [Figure 1-1](#), the icons look similar.



Figure 1-1. R and RStudio icons in macOS

If you click on the R icon, you'll be greeted by something like [Figure 1-2](#), which is the Base R interface on a Mac, but certainly not RStudio.

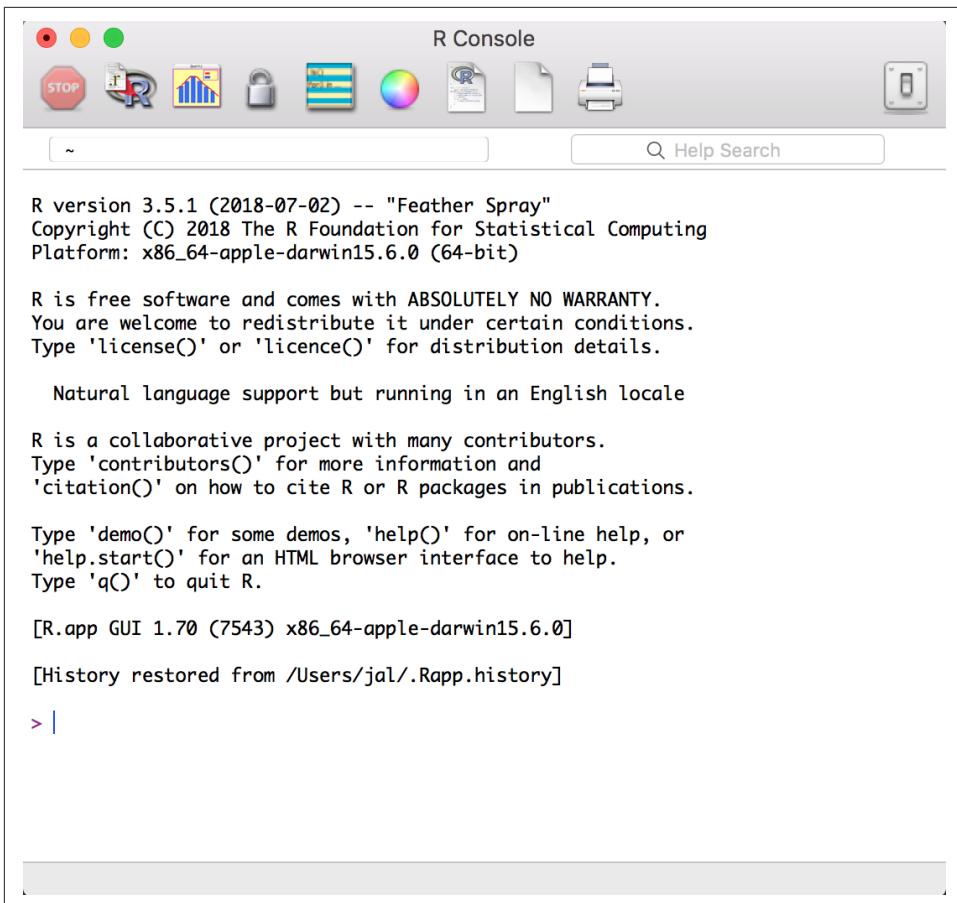


Figure 1-2. The R console in macOS

When you start RStudio, by default it will reopen the last project you were working on in RStudio.

1.4 Entering Commands

Problem

You've started RStudio. Now what?

Solution

When you start RStudio, the main window on the left is an R session. From there you can enter commands interactively directly to R.

Discussion

R prompts you with `>`. To get started, just treat R like a big calculator: enter an expression, and R will evaluate the expression and print the result:

```
> 1 + 1  
[1] 2  
>
```

The computer adds 1 and 1, and displays the result, 2.

The `[1]` before the 2 might be confusing. To R, the result is a vector, even though it has only one element. R labels the value with `[1]` to signify that this is the first element of the vector... which is not surprising, since it's the *only* element of the vector.

R will prompt you for input until you type a complete expression. The expression `max(1,3,5)` is a complete expression, so R stops reading input and evaluates what it's got:

```
> max(1, 3, 5)  
[1] 5  
>
```

In contrast, `max(1,3,` is an incomplete expression, so R prompts you for more input. The prompt changes from greater-than (`>`) to plus (`+`), letting you know that R expects more:

```
> max(1, 3,  
+ 5)  
[1] 5  
>
```

It's easy to mistype commands, and retyping them is tedious and frustrating. So R includes command-line editing to make life easier. It defines single keystrokes that let you easily recall, correct, and reexecute your commands. A typical command-line interaction goes like this:

1. You enter an R expression with a typo.
2. R complains about your mistake.
3. You press the up arrow key to recall your mistaken line.
4. You use the left and right arrow keys to move the cursor back to the error.
5. You use the Delete key to delete the offending characters.
6. You type the corrected characters, which inserts them into the command line.

7. You press Enter to reexecute the corrected command.

That's just the basics. R supports the usual keystrokes for recalling and editing command lines, as listed in [Table 1-2](#).

Table 1-2. R command shortcuts

Labeled key	Ctrl-key combo	Effect
Up arrow	Ctrl-P	Recall previous command by moving backward through the history of commands.
Down arrow	Ctrl-N	Move forward through the history of commands.
Backspace	Ctrl-H	Delete the character to the left of the cursor.
Delete (Del)	Ctrl-D	Delete the character to the right of the cursor.
Home	Ctrl-A	Move the cursor to the start of the line.
End	Ctrl-E	Move the cursor to the end of the line.
Right arrow	Ctrl-F	Move the cursor right (forward) one character.
Left arrow	Ctrl-B	Move the cursor left (back) one character.
	Ctrl-K	Delete everything from the cursor position to the end of the line.
	Ctrl-U	Clear the whole darn line and start over.
Tab		Complete the name (on some platforms).

On most operating systems, you can also use the mouse to highlight commands and then use the usual copy and paste commands to paste text into a new command line.

See Also

See [Recipe 2.12](#). From the Windows main menu, follow Help → Console for a complete list of keystrokes useful for command-line editing.

1.5 Exiting from RStudio

Problem

You want to exit from RStudio.

Solution

Windows and most Linux distributions

Select File → Quit Session from the main menu, or click on the X in the upper-right corner of the window frame.

macOS

Select File → Quit Session from the main menu, or press Cmd-Q, or click on the red circle in the upper-left corner of the window frame.

On all platforms, you can also use the `q` function (as in `quit`) to terminate R and RStudio:

`q()`

Note the empty parentheses, which are necessary to call the function.

Discussion

Whenever you exit, R typically asks if you want to save your workspace. You have three choices:

- Save your workspace and exit.
- Don't save your workspace, but exit anyway.
- Cancel, returning to the command prompt rather than exiting.

If you save your workspace, R writes it to a file called `.RData` in the current working directory. Saving the workspace saves any R objects you have created. The next time you start R in the same directory, the workspace will automatically load. Saving your workspace will overwrite the previously saved workspace, if any, so don't save if you don't like your changes (e.g., if you have accidentally erased critical data from your workspace).

We recommend never saving your workspace when you exit and instead always explicitly saving your project, scripts, and data. We also recommend that you turn off the prompt to save and autorestore the workspace in RStudio using the global options found in the menu Tools → Global Options and shown in [Figure 1-3](#). This way, when you exit R and RStudio, you won't be prompted to save your workspace. But keep in mind that any objects created but not saved to disk will be lost!

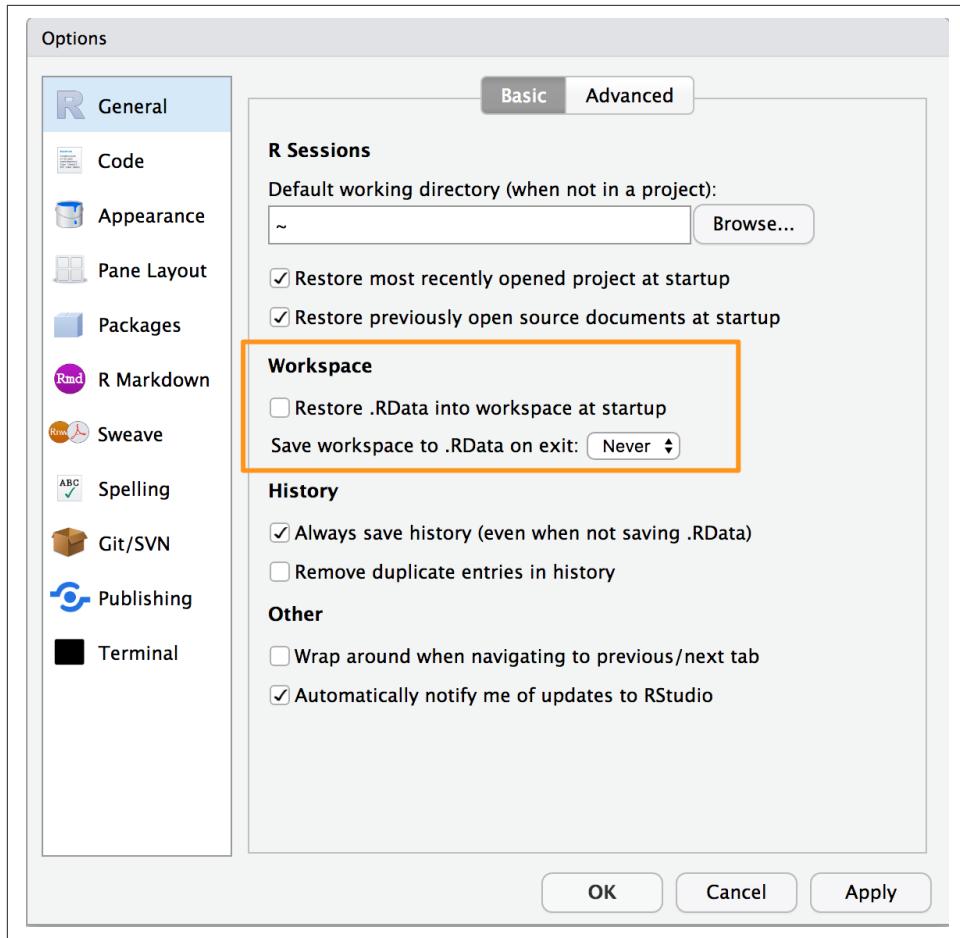


Figure 1-3. Save workspace options

See Also

See [Recipe 3.1](#) for more about the current working directory and [Recipe 3.3](#) for more about saving your workspace. Also see Chapter 2 of *R in a Nutshell*.

1.6 Interrupting R

Problem

You want to interrupt a long-running computation and return to the command prompt without exiting RStudio.

Solution

Press the Esc key on your keyboard, or click on the Session menu in RStudio and select “Interrupt R.” You may also click on the stop sign icon in the code console window.

Discussion

Interrupting R means telling R to stop running the current command, but without deleting variables from memory or completely closing RStudio. That said, interrupting R can leave your variables in an indeterminate state, depending upon how far the computation had progressed, so check your workspace after interrupting.

See Also

See [Recipe 1.5](#).

1.7 Viewing the Supplied Documentation

Problem

You want to read the documentation supplied with R.

Solution

Use the `help.start` function to see the documentation’s table of contents:

```
help.start()
```

From there, links are available to all the installed documentation. In RStudio the help will show up in the help pane, which by default is on the righthand side of the screen.

In RStudio you can also click Help → R Help to get a listing with help options for both R and RStudio.

Discussion

The base distribution of R includes a wealth of documentation—literally thousands of pages. When you install additional packages, those packages contain documentation that is also installed on your machine.

It is easy to browse this documentation via the `help.start` function, which opens on the top-level table of contents. [Figure 1-4](#) shows how `help.start` appears inside the help pane in RStudio.

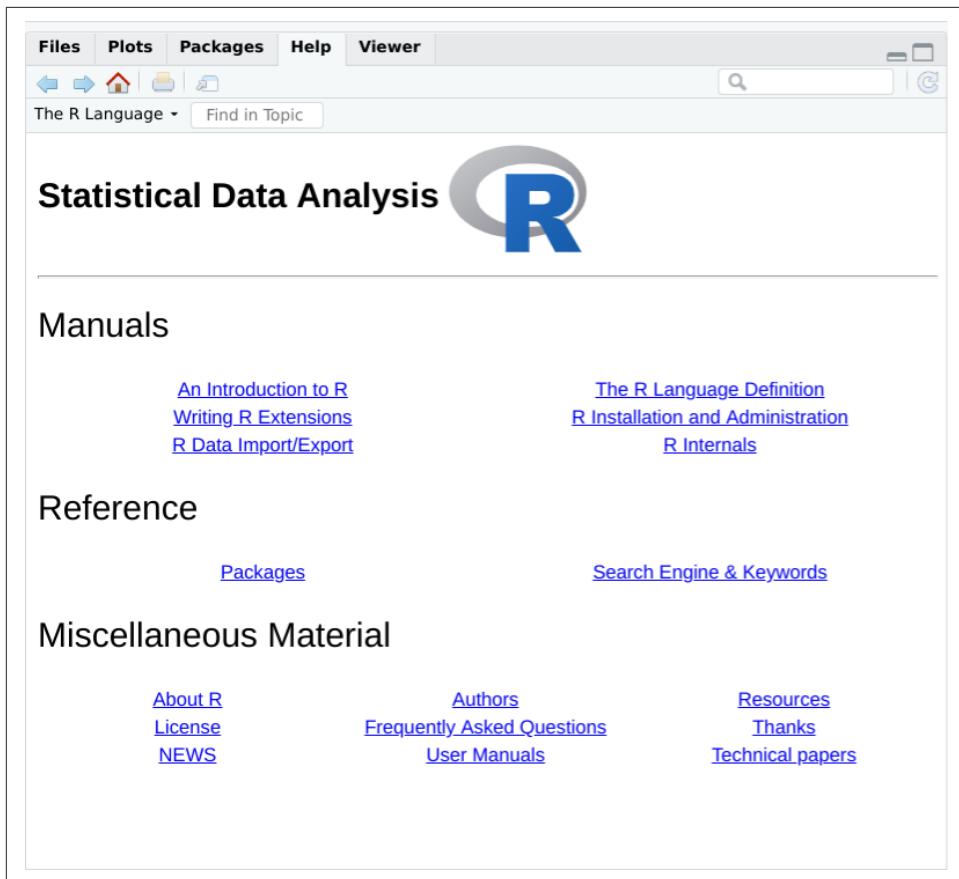


Figure 1-4. RStudio help.start

The two links in the Reference section are especially useful:

Packages

Click here to see a list of all the installed packages—both the base packages and the additional installed packages. Click on a package name to see a list of its functions and datasets.

Search Engine & Keywords

Click here to access a simple search engine that allows you to search the documentation by keyword or phrase. There is also a list of common keywords, organized by topic; click one to see the associated pages.

The Base R documentation accessed via `help.start` is loaded on your computer when you install R. The RStudio help, which you access by using the menu option

Help → R Help, presents a page with links to RStudio’s website. So, you will need internet access to access the RStudio help links.

See Also

The local documentation is copied from the [R Project website](#), which may have updated documents.

1.8 Getting Help on a Function

Problem

You want to know more about a function that is installed on your machine.

Solution

Use `help` to display the documentation for the function:

```
help(functionname)
```

Use `args` for a quick reminder of the function arguments:

```
args(functionname)
```

Use `example` to see examples of using the function:

```
example(functionname)
```

Discussion

We present many R functions in this book. Every R function has more bells and whistles than we can possibly describe. If a function catches your interest, we strongly suggest reading the help page for that function. One of its bells or whistles might be very useful to you.

Suppose you want to know more about the `mean` function. Use the `help` function like this:

```
help(mean)
```

This will open the help page for the `mean` function in the help pane in RStudio. A shortcut for the `help` command is to simply type `?` followed by the function name:

```
?mean
```

Sometimes you just want a quick reminder of the arguments to a function: what are they, and in what order do they occur? For this case, use the `args` function:

```
args(mean)
#> function (x, ...)
#> NULL

args(sd)
#> function (x, na.rm = FALSE)
#> NULL
```

The first line of output from `args` is a synopsis of the function call. For `mean`, the synopsis shows one argument, `x`, which is a vector of numbers. For `sd`, the synopsis shows the same vector, `x`, and an optional argument called `na.rm`. (You can ignore the second line of output, which is often just `NULL`.) In RStudio you will see the `args` output as a floating tool tip over your cursor when you type a function name, as shown in Figure 1-5.

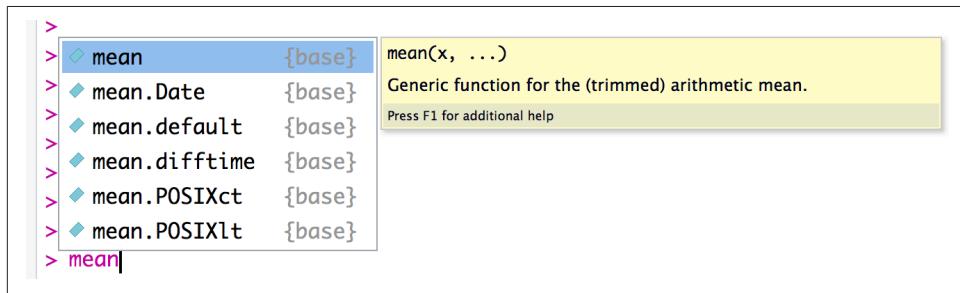


Figure 1-5. RStudio tool tip

Most documentation for functions includes example code near the end of the document. A cool feature of R is that you can request that it execute the examples, giving you a little demonstration of the function's capabilities. The documentation for the `mean` function, for instance, contains examples, but you don't need to type them yourself. Just use the `example` function to watch them run:

```
example(mean)
#>
#> mean> x <- c(0:10, 50)
#>
#> mean> xm <- mean(x)
#>
#> mean> c(xm, mean(x, trim = 0.10))
#> [1] 8.75 5.50
```

Everything you see after `example(mean)` was produced by R, which executed the examples from the help page and displayed the results.

See Also

See [Recipe 1.9](#) for searching for functions and [Recipe 3.6](#) for more about the search path.

1.9 Searching the Supplied Documentation

Problem

You want to know more about a function that is installed on your machine, but the `help` function reports that it cannot find documentation for any such function.

Alternatively, you want to search the installed documentation for a keyword.

Solution

Use `help.search` to search the R documentation on your computer:

```
help.search("pattern")
```

A typical pattern is a function name or keyword. Notice that it must be enclosed in quotation marks.

For your convenience, you can also invoke a search by using two question marks (in which case the quotes are not required). Note that searching for a function by name uses one question mark, while searching for a text pattern uses two:

```
> ??pattern
```

Discussion

You may occasionally request help on a function only to be told R knows nothing about it:

```
help(adf.test)
#> No documentation for 'adf.test' in specified packages and libraries:
#> you could try '??adf.test'
```

This can be frustrating if you *know* the function is installed on your machine. Here the problem is that the function's package is not currently loaded, and you don't know which package contains the function. It's kind of a catch-22 (the error message indicates the package is not currently in your search path, so R cannot find the help file; see [Recipe 3.6](#) for more details).

The solution is to search all your installed packages for the function. Just use the `help.search` function, as suggested in the error message:

```
help.search("adf.test")
```

The search will produce a listing of all packages that contain the function:

```
Help files with alias or concept or title matching 'adf.test' using
regular expression matching:

tseries::adf.test      Augmented Dickey-Fuller Test
```

```
Type '?PKG::FOO' to inspect entry 'PKG::FOO TITLE'.
```

The preceding output indicates that the `tseries` package contains the `adf.test` function. You can see its documentation by explicitly telling `help` which package contains the function:

```
help(adf.test, package = "tseries")
```

or you can use the double colon operator to tell R to look in a specific package:

```
?tseries::adf.test
```

You can broaden your search by using keywords. R will then find any installed documentation that contains the keywords. Suppose you want to find all functions that mention the Augmented Dickey–Fuller (ADF) test. You could search on a likely pattern:

```
help.search("dickey-fuller")
```

See Also

You can also access the local search engine through the documentation browser; see [Recipe 1.7](#) for how this is done. See [Recipe 3.6](#) for more about the search path and [Recipe 1.8](#) for getting help on functions.

1.10 Getting Help on a Package

Problem

You want to learn more about a package installed on your computer.

Solution

Use the `help` function and specify a package name (without a function name):

```
help(package = "packagename")
```

Discussion

Sometimes you want to know the contents of a package (the functions and datasets). This is especially true after you download and install a new package, for example. The `help` function can provide the contents plus other information once you specify the package name.

This call to `help` would display the information for the `tseries` package, a standard package in the base distribution (try it!):

```
help(package = "tseries")
```

The information begins with a description and continues with an index of functions and datasets. In RStudio, the HTML-formatted help page will open in the help window of the IDE.

Some packages also include vignettes, which are additional documents such as introductions, tutorials, or reference cards. They are installed on your computer as part of the package documentation when you install the package. The help page for a package includes a list of its vignettes near the bottom.

You can see a list of all vignettes on your computer by using the `vignette` function:

```
vignette()
```

In RStudio this will open a new tab listing every package installed on your computer that includes vignettes as well as the vignette names and descriptions.

You can see the vignettes for a particular package by including its name:

```
vignette(package = "packagename")
```

Each vignette has a name, which you use to view the vignette:

```
vignette("vignettename")
```

See Also

See [Recipe 1.8](#) for getting help on a particular function in a package.

1.11 Searching the Web for Help

Problem

You want to search the web for information and answers regarding R.

Solution

Inside R, use the `RSiteSearch` function to search by keyword or phrase:

```
RSiteSearch("key phrase")
```

Inside your browser, try using these sites for searching:

RSeek

This is a Google custom search engine that is focused on R-specific websites.

Stack Overflow

Stack Overflow is a searchable Q&A site from Stack Exchange that is oriented toward programming issues such as data structures, coding, and graphics. Stack Overflow is a great “first stop” for all your syntax questions.

Cross Validated

Cross Validated is a Stack Exchange site focused on statistics, machine learning, and data analysis rather than programming. It's a good place for questions about what statistical method to use.

RStudio Community

The RStudio Community site is a discussion forum hosted by RStudio. The topics include R, RStudio, and associated technology. Being an RStudio site, this forum is often visited by RStudio staff and those who use the software frequently. This is a good place for general questions and questions that possibly don't fit as well into the Stack Overflow syntax-focused format.

Discussion

The `RSiteSearch` function will open a browser window and direct it to the search engine on the [R Project website](#). There you will see an initial search that you can refine. For example, this call would start a search for “canonical correlation”:

```
RSiteSearch("canonical correlation")
```

This is quite handy for doing quick web searches without leaving R. However, the search scope is limited to R documentation and the mailing list archives.

[RSeek](#) provides a wider search. Its virtue is that it harnesses the power of the Google search engine while focusing on sites relevant to R. That eliminates the extraneous results of a generic Google search. The beauty of RSeek is that it organizes the results in a useful way.

Figure 1-6 shows the results of visiting RSeek and searching for “correlation.” Note that the tabs across the top allow for drilling in to different types of content:

- All results
- Packages
- Books
- Support
- Articles
- For Beginners

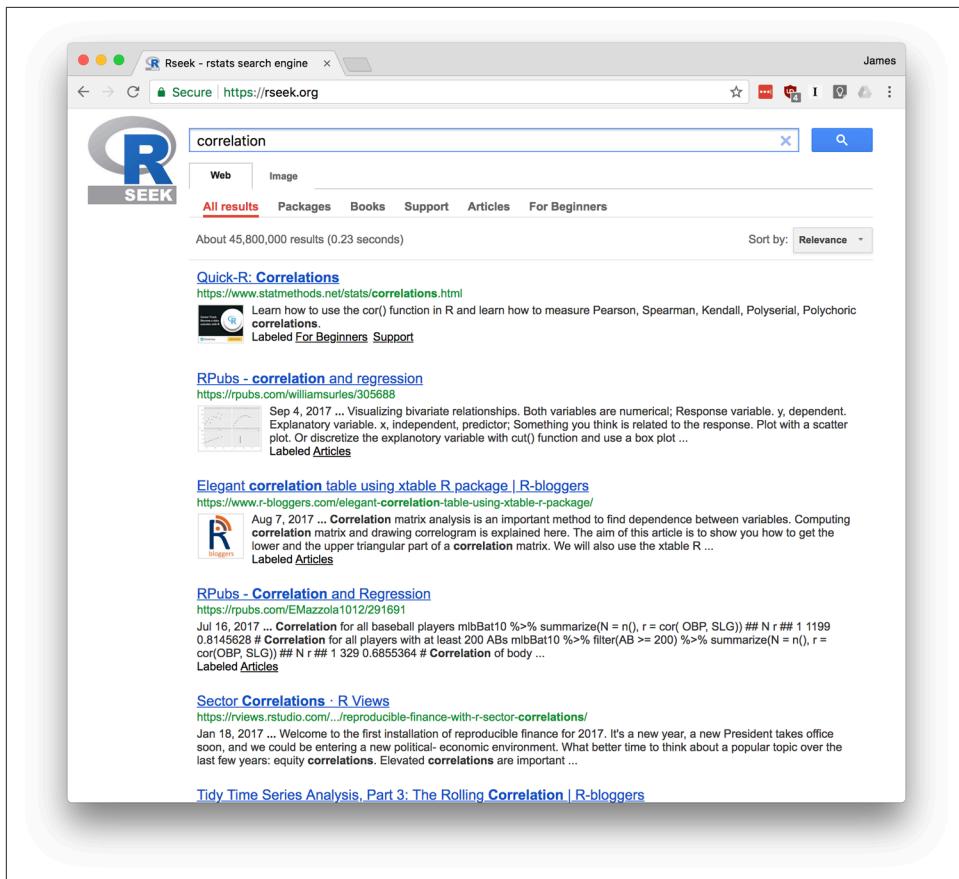


Figure 1-6. RSeek

Stack Overflow is a Q&A site, which means that anyone can submit a question and experienced users will supply answers—often there are multiple answers to each question. Readers vote on the answers, so good answers tend to rise to the top. This creates a rich database of Q&A dialogues, which you can search. Stack Overflow is strongly problem-oriented, and the topics lean toward the programming side of R.

Stack Overflow hosts questions for many programming languages; therefore, when entering a term into its search box, prefix it with “[r]” to focus the search on questions tagged for R. For example, searching for “[r] standard error” will select only the questions tagged for R and will avoid the Python and C++ questions.

Stack Overflow also includes a [wiki about the R language](#) that provides an excellent community-curated list of online R resources.

Stack Exchange (the parent company of Stack Overflow) has a Q&A area for statistical analysis called **Cross Validated**. This area is more focused on statistics than pro-

gramming, so use it when seeking answers that are more concerned with statistics in general and less with R in particular.

RStudio hosts its own [discussion board](#) as well. This is a great place to ask general questions and more conceptual questions that may not work as well on Stack Overflow.

See Also

If your search reveals a useful package, use [Recipe 3.10](#) to install it on your machine.

1.12 Finding Relevant Functions and Packages

Problem

Of the 10,000+ packages for R, you have no idea which ones would be useful to you.

Solution

- To discover packages related to a certain field, visit CRAN’s [list of task views](#). Select the task view for your area, which will give you links to and descriptions of relevant packages. Or visit [RSeek](#), search by keyword, click on the Task Views tab, and select an applicable task view.
- Visit [crantastic](#) and search for packages by keyword.
- To find relevant functions, visit [RSeek](#), search by name or keyword, and click on the Functions tab.

Discussion

This problem is especially vexing for beginners. You think R can solve your problems, but you have no idea which packages and functions would be useful. A common question on the mailing lists is: “Is there a package to solve problem X?” That is the silent scream of someone drowning in R.

As of this writing, there are more than 10,000 packages available for free download from CRAN. Each package has a summary page with a short description and links to the package documentation. Once you’ve located a potentially interesting package, you would typically click on the “Reference manual” link to view the PDF documentation with full details. (The summary page also contains download links for installing the package, but you’ll rarely install the package that way; see [Recipe 3.10](#).)

Sometimes you simply have a generic interest—such as Bayesian analysis, econometrics, optimization, or graphics. CRAN contains a set of task view pages describing

packages that may be useful. A task view is a great place to start since you get an overview of what's available. You can see the list of task view pages at [CRAN Task Views](#) or search for them as described in the Solution. CRAN's Task Views lists a number of broad fields and shows packages that are used in each field. For example, there are task views for high-performance computing, genetics, time series, and social science, just to name a few.

Suppose you happen to know the name of a useful package—say, by seeing it mentioned online. A complete alphabetical list of packages is available at [CRAN](#) with links to the package summary pages.

See Also

You can download and install an R package called `sos` that provides powerful other ways to search for packages; see the [vignette at SOS](#).

1.13 Searching the Mailing Lists

Problem

You have a question, and you want to search the archives of the mailing lists to see whether your question was answered previously.

Solution

Open [Nabble](#) in your browser. Search for a keyword or other search term from your question. This will show results from the support mailing lists.

Discussion

This recipe is really just an application of [Recipe 1.11](#). But it's an important application, because you should search the mailing list archives before submitting a new question to the list. Your question has probably been answered before.

See Also

CRAN has a list of additional resources for searching the web; see [CRAN Search](#).

1.14 Submitting Questions to Stack Overflow or Elsewhere in the Community

Problem

You have a question you can't find the answer to online, so you want to submit a question to the R community.

Solution

The first step to asking a question online is to create a reproducible example. Having example code that someone can run and see your exact problem is the most critical part of asking for help online. A question with a good reproducible example has three components:

Example data

This can be simulated data or some real data that you provide.

Example code

This code shows what you have tried or an error you are getting.

Written description

This is where you explain what you have, what you'd like to have, and what you have tried that didn't work.

The details of writing a reproducible example are covered in the Discussion. Once you have a reproducible example, you can post your question on [Stack Overflow](#). Be sure to include the `r` tag in the Tags section of the ask page.

If your question is more general or related to concepts instead of specific syntax, RStudio runs an RStudio Community [discussion forum](#). Note that the site is broken into multiple topics, so pick the topic category that best fits your question.

Or you may submit your question to the R mailing lists (but don't submit to multiple sites, the mailing lists, and Stack Overflow, as that's considered rude cross-posting).

The [mailing lists page](#) contains general information and instructions for using the R-help mailing list. Here is the general process:

1. Subscribe to the main R mailing list, [R-help](#).
2. Write your question carefully and correctly and include your reproducible example.
3. Mail your question to `r-help@r-project.org`.

Discussion

The R-help mailing list, Stack Overflow, and the RStudio Community site are great resources, but please treat them as a last resort. Read the help pages, read the documentation, search the help list archives, and search the web. It is most likely that your question has already been answered. Don't kid yourself: very few questions are unique. If you've exhausted all other options, though, maybe it's time to create a good question.

The reproducible example is the crux of a good help request. The first component is example data. A good way to get this is to simulate the data using a few R functions. The following example creates a data frame called `example_df` that has three columns, each of a different data type:

```
set.seed(42)
n <- 4
example_df <- data.frame(
  some_reals = rnorm(n),
  some_letters = sample(LETTERS, n, replace = TRUE),
  some_ints = sample(1:10, n, replace = TRUE)
)
example_df
#>   some_reals some_letters some_ints
#> 1      1.371          R         10
#> 2     -0.565          S          3
#> 3      0.363          L          5
#> 4      0.633          S         10
```

Note that this example uses the command `set.seed` at the beginning. This ensures that every time this code is run, the answers will be the same. The `n` value is the number of rows of example data you would like to create. Make your example data as simple as possible to illustrate your question.

An alternative to creating simulated data is to use example data that comes with R. For example, the dataset `mtcars` contains a data frame with 32 records about different car models:

```
data(mtcars)
head(mtcars)
#>           mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4     21.0   6 160 110 3.90 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag 21.0   6 160 110 3.90 2.88 17.0  0  1    4    4
#> Datsun 710    22.8   4 108  93 3.85 2.32 18.6  1  1    4    1
#> Hornet 4 Drive 21.4   6 258 110 3.08 3.21 19.4  1  0    3    1
#> Hornet Sportabout 18.7   8 360 175 3.15 3.44 17.0  0  0    3    2
#> Valiant       18.1   6 225 105 2.76 3.46 20.2  1  0    3    1
```

If your example is reproducible only with your own data, you can use `dput` to put a bit of your own data in a string that you can use in your example. We'll illustrate that approach using two rows from the `mtcars` dataset:

```
dput(head(mtcars, 2))
#> structure(list(mpg = c(21, 21), cyl = c(6, 6), disp = c(160,
#> 160), hp = c(110, 110), drat = c(3.9, 3.9), wt = c(2.62, 2.875
#> ), qsec = c(16.46, 17.02), vs = c(0, 0), am = c(1, 1), gear = c(4,
#> 4), carb = c(4, 4)), row.names = c("Mazda RX4", "Mazda RX4 Wag"
#> ), class = "data.frame")
```

You can put the resulting structure directly in your question:

```
example_df <- structure(list(mpg = c(21, 21), cyl = c(6, 6), disp = c(160,
160), hp = c(110, 110), drat = c(3.9, 3.9), wt = c(2.62, 2.875
), qsec = c(16.46, 17.02), vs = c(0, 0), am = c(1, 1), gear = c(4,
4), carb = c(4, 4)), row.names = c("Mazda RX4", "Mazda RX4 Wag"
), class = "data.frame")

example_df
#>          mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4     21   6  160 110  3.9 2.62 16.5   0  1    4    4
#> Mazda RX4 Wag 21   6  160 110  3.9 2.88 17.0   0  1    4    4
```

The second part of a good reproducible example is the example code. The code example should be as simple as possible and illustrate what you are trying to do or have already tried. It should *not* be a big block of code with many different things going on. Boil your example down to only the minimal amount of code needed. If you use any packages, be sure to include the `library` call at the beginning of your code. Also, don't include anything in your question that is potentially harmful to someone running your code, such as `rm(list=ls())`, which would delete all R objects in memory. Have empathy for the person trying to help you, and realize that they are volunteering their time to help you out and may run your code on the same machine they use to do their own work.

To test your example, open a new R session and try running it. Once you've edited your code, it's time to give just a bit more information to your potential respondents. In plain text, describe what you were trying to do, what you've tried, and your question. Be as concise as possible. As with the example code, your objective is to communicate as efficiently as possible with the person reading your question. You may find it helpful to include in your description which version of R you are running as well as which platform (Windows, Mac, Linux). You can get that information easily with the `sessionInfo` command.

If you are going to submit your question to the R mailing list, you should know there are actually several mailing lists. R-help is the main list for general questions. There are also many special interest group (SIG) mailing lists dedicated to particular domains such as genetics, finance, R development, and even R jobs. You can see the full list at <https://stat.ethz.ch/mailman/listinfo>. If your question is specific to a domain, you'll get a better answer by selecting the appropriate list. As with R-help, however, carefully search the SIG list archives before submitting your question.

See Also

We suggest that you read Eric Raymond and Rick Moen's excellent essay entitled "[How to Ask Questions the Smart Way](#)" before submitting any question. Seriously. Read it.

Stack Overflow has an excellent post that includes details about creating a reproducible example. You can find that at <https://stackoverflow.com/q/5963269/37751>.

Jenny Bryan has a great R package called `reprex` that helps in the creation of a good reproducible example and provides helper functions for writing the markdown text for sites like Stack Overflow. You can find that package on her [GitHub page](#).

CHAPTER 2

Some Basics

The recipes in this chapter lie somewhere between problem-solving ideas and tutorials. Yes, they solve common problems, but the Solutions showcase common techniques and idioms used in most R code, including the code in this cookbook. If you are new to R, we suggest skimming this chapter to acquaint yourself with these idioms.

2.1 Printing Something to the Screen

Problem

You want to display the value of a variable or expression.

Solution

If you simply enter the variable name or expression at the command prompt, R will print its value. Use the `print` function for generic printing of any object. Use the `cat` function for producing custom-formatted output.

Discussion

It's very easy to ask R to print something—just enter it at the command prompt:

```
pi  
#> [1] 3.14  
sqrt(2)  
#> [1] 1.41
```

When you enter expressions like these, R evaluates the expression and then implicitly calls the `print` function. So the previous example is identical to this:

```
print(pi)
#> [1] 3.14
print(sqrt(2))
#> [1] 1.41
```

The beauty of `print` is that it knows how to format any R value for printing, including structured values such as matrices and lists:

```
print(matrix(c(1, 2, 3, 4), 2, 2))
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
print(list("a", "b", "c"))
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> [1] "b"
#>
#> [[3]]
#> [1] "c"
```

This is useful because you can always view your data: just `print` it. You need not write special printing logic, even for complicated data structures.

The `print` function has a significant limitation, however: it prints only one object at a time. Trying to `print` multiple items gives this mind-numbing error message:

```
print("The zero occurs at", 2 * pi, "radians.")
#> Error in print.default("The zero occurs at", 2 * pi, "radians."):
#>   invalid 'quote' argument
```

The only way to `print` multiple items is to print them one at a time, which probably isn't what you want:

```
print("The zero occurs at")
#> [1] "The zero occurs at"
print(2 * pi)
#> [1] 6.28
print("radians")
#> [1] "radians"
```

The `cat` function is an alternative to `print` that lets you concatenate multiple items into a continuous output:

```
cat("The zero occurs at", 2 * pi, "radians.", "\n")
#> The zero occurs at 6.28 radians.
```

Notice that `cat` puts a space between each item by default. You must provide a new-line character (`\n`) to terminate the line.

The `cat` function can print simple vectors, too:

```
fib <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
cat("The first few Fibonacci numbers are:", fib, "...\\n")
#> The first few Fibonacci numbers are: 0 1 1 2 3 5 8 13 21 34 ...
```

Using `cat` gives you more control over your output, which makes it especially useful in R scripts that generate output consumed by others. A serious limitation, however, is that it cannot print compound data structures such as matrices and lists. Trying to `cat` them only produces another mind-numbing message:

```
cat(list("a", "b", "c"))
#> Error in cat(list("a", "b", "c")): argument 1 (type 'list') cannot
#>     be handled by 'cat'
```

See Also

See [Recipe 4.2](#) for controlling output format.

2.2 Setting Variables

Problem

You want to save a value in a variable.

Solution

Use the assignment operator (`<-`). There is no need to declare your variable first:

```
x <- 3
```

Discussion

Using R in “calculator mode” gets old pretty fast. Soon you will want to define variables and save values in them. This reduces typing, saves time, and clarifies your work.

There is no need to declare or explicitly create variables in R. Just assign a value to the name and R will create the variable:

```
x <- 3
y <- 4
z <- sqrt(x^2 + y^2)
print(z)
#> [1] 5
```

Notice that the assignment operator is formed from a less-than character (`<`) and a hyphen (`-`) with no space between them.

When you define a variable at the command prompt like this, the variable is held in your workspace. The workspace is held in the computer's main memory but can be saved to disk. The variable definition remains in the workspace until you remove it.

R is a *dynamically typed language*, which means that we can change a variable's data type at will. We could set `x` to be numeric, as just shown, and then turn around and immediately overwrite that with (say) a vector of character strings. R will not complain:

```
x <- 3
print(x)
#> [1] 3

x <- c("fee", "fie", "foe", "fum")
print(x)
#> [1] "fee" "fie" "foe" "fum"
```

In some R functions you will see assignment statements that use the strange-looking assignment operator `<<-`:

```
x <<- 3
```

That forces the assignment to a global variable rather than a local variable. Scoping is a bit, well, out of scope for this discussion, however.

In the spirit of full disclosure, we will reveal that R also supports two other forms of assignment statements. A single equals sign (`=`) can be used as an assignment operator. A rightward assignment operator (`->`) can be used anywhere the leftward assignment operator (`<-`) can be used (but with the arguments reversed):

```
foo <- 3
print(foo)
#> [1] 3

5 -> fum
print(fum)
#> [1] 5
```

We recommend that you avoid these as well. The equals-sign assignment is easily confused with the test for equality. The rightward assignment can be useful in certain contexts, but it can be confusing to those not used to seeing it.

See Also

See Recipes 2.4, 2.14, and 3.3. See also the help page for the `assign` function.

2.3 Listing Variables

Problem

You want to know what variables and functions are defined in your workspace.

Solution

Use the `ls` function. Use `ls.str` for more details about each variable. You can also see your variables and functions in the Environment pane in RStudio, shown in the next recipe in [Figure 2-1](#).

Discussion

The `ls` function displays the names of objects in your workspace:

```
x <- 10
y <- 50
z <- c("three", "blind", "mice")
f <- function(n, p) sqrt(p * (1 - p) / n)
ls()
#> [1] "f"  "x"  "y"  "z"
```

Notice that `ls` returns a vector of character strings in which each string is the name of one variable or function. When your workspace is empty, `ls` returns an empty vector, which produces this puzzling output:

```
ls()
#> character(0)
```

That is R's quaint way of saying that `ls` returned a zero-length vector of strings; that is, it returned an empty vector because nothing is defined in your workspace.

If you want more than just a list of names, try `ls.str`; this will also tell you something about each variable:

```
x <- 10
y <- 50
z <- c("three", "blind", "mice")
f <- function(n, p) sqrt(p * (1 - p) / n)
ls.str()
#> f : function (n, p)
#> x : num 10
#> y : num 50
#> z : chr [1:3] "three" "blind" "mice"
```

The function is called `ls.str` because it is both listing your variables and applying the `str` function to them, showing their structure (see [Recipe 12.13](#)).

Ordinarily, `ls` does not return any name that begins with a dot (.). Such names are considered hidden and are not normally of interest to users. (This mirrors the Unix convention of not listing files whose names begin with a dot.) You can force `ls` to list everything by setting the `all.names` argument to TRUE:

```
ls()  
#> [1] "f" "x" "y" "z"  
ls(all.names = TRUE)  
#> [1] ".Random.seed" "f" "x" "y"  
#> [5] "z"
```

The Environment pane in RStudio also hides objects with names that begin with a dot.

See Also

See [Recipe 2.4](#) for deleting variables and [Recipe 12.13](#) for inspecting your variables.

2.4 Deleting Variables

Problem

You want to remove unneeded variables or functions from your workspace or to erase its contents completely.

Solution

Use the `rm` function.

Discussion

Your workspace can get cluttered quickly. The `rm` function removes, permanently, one or more objects from the workspace:

```
x <- 2 * pi  
x  
#> [1] 6.28  
rm(x)  
x  
#> Error in eval(expr, envir, enclos): object 'x' not found
```

There is no “undo”; once the variable is gone, it’s gone.

You can remove several variables at once:

```
rm(x, y, z)
```

You can even erase your entire workspace at once. The `rm` function has a `list` argument consisting of a vector of names of variables to remove. Recall that the `ls`

function returns a vector of variable names; hence, you can combine `rm` and `ls` to erase everything:

```
ls()  
#> [1] "f"  "x"  "y"  "z"  
rm(list = ls())  
ls()  
#> character(0)
```

Alternatively, you could click the broom icon at the top of the Environment pane in RStudio, shown in [Figure 2-1](#).

The screenshot shows the RStudio Environment pane. At the top, there are tabs for Environment, History, and Connections, with Environment selected. Below the tabs are icons for file operations (New File, Save, Import Dataset), a search bar, and a 'List' dropdown set to 'List'. The main area is titled 'Global Environment'. It contains sections for 'Data', 'Values', and 'Functions'. Under 'Data', there are two entries: 'dat' (100 obs. of 1 variable) and 'data_to_plot' (26 obs. of 3 variables). Under 'Values', there are three entries: 'x' (10), 'y' (50), and 'z' (chr [1:3] "three" "blind" "mice"). Under 'Functions', there is one entry: 'f' (function (n, p)).

Figure 2-1. Environment pane in RStudio



Never put `rm(list=ls())` into code you share with others, such as a library function or sample code sent to a mailing list or Stack Overflow. Deleting all the variables in someone else's workspace is worse than rude and will make you extremely unpopular.

See Also

See [Recipe 2.3](#).

2.5 Creating a Vector

Problem

You want to create a vector.

Solution

Use the `c(...)` operator to construct a vector from given values.

Discussion

Vectors are a central component of R, not just another data structure. A vector can contain either numbers, strings, or logical values, but not a mixture.

The `c(...)` operator can construct a vector from simple elements:

```
c(1, 1, 2, 3, 5, 8, 13, 21)
#> [1] 1 1 2 3 5 8 13 21
c(1 * pi, 2 * pi, 3 * pi, 4 * pi)
#> [1] 3.14 6.28 9.42 12.57
c("My", "twitter", "handle", "is", "@cmastication")
#> [1] "My"           "twitter"        "handle"        "is"
#> [5] "@cmastication"
c(TRUE, TRUE, FALSE, TRUE)
#> [1] TRUE TRUE FALSE TRUE
```

If the arguments to `c(...)` are themselves vectors, it flattens them and combines them into one single vector:

```
v1 <- c(1, 2, 3)
v2 <- c(4, 5, 6)
c(v1, v2)
#> [1] 1 2 3 4 5 6
```

Vectors cannot contain a mix of data types, such as numbers and strings. If you create a vector from mixed elements, R will try to accommodate you by converting one of them:

```
v1 <- c(1, 2, 3)
v3 <- c("A", "B", "C")
c(v1, v3)
#> [1] "1" "2" "3" "A" "B" "C"
```

Here, we tried to create a vector from both numbers and strings. R converted all the numbers to strings before creating the vector, thereby making the data elements compatible. Note that R does this without warning or complaint.

Technically speaking, two data elements can coexist in a vector only if they have the same *mode*. The modes of 3.1415 and "foo" are `numeric` and `character`, respectively:

```
mode(3.1415)
#> [1] "numeric"
mode("foo")
#> [1] "character"
```

Those modes are incompatible. To make a vector from them, R converts 3.1415 to `character` mode so it will be compatible with "foo":

```
c(3.1415, "foo")
#> [1] "3.1415" "foo"
mode(c(3.1415, "foo"))
#> [1] "character"
```



`c` is a generic operator, which means that it works with many data types and not just vectors. However, it might not do exactly what you expect, so check its behavior before applying it to other data types and objects.

See Also

See the introduction to [Chapter 5](#) for more about vectors and other data structures.

2.6 Computing Basic Statistics

Problem

You want to calculate basic statistics: mean, median, standard deviation, variance, correlation, or covariance.

Solution

Use one of these functions, assuming that `x` and `y` are vectors:

- `mean(x)`
- `median(x)`
- `sd(x)`
- `var(x)`
- `cor(x, y)`
- `cov(x, y)`

Discussion

When you first use R you might open the documentation and begin searching for material entitled “Procedures for Calculating Standard Deviation.” It seems that such an important topic would likely require a whole chapter.

It’s not that complicated.

Standard deviation and other basic statistics are calculated by simple functions. Ordinarily, the function argument is a vector of numbers and the function returns the calculated statistic:

```
x <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
mean(x)
#> [1] 8.8
median(x)
#> [1] 4
sd(x)
#> [1] 11
var(x)
#> [1] 122
```

The `sd` function calculates the sample standard deviation, and `var` calculates the sample variance.

The `cor` and `cov` functions can calculate the correlation and covariance, respectively, between two vectors:

```
x <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
y <- log(x + 1)
cor(x, y)
#> [1] 0.907
cov(x, y)
#> [1] 11.5
```

All these functions are picky about values that are not available (`NA`). Even one `NA` value in the vector argument causes any of these functions to return `NA` or even halt altogether with a cryptic error:

```
x <- c(0, 1, 1, 2, 3, NA)
mean(x)
#> [1] NA
sd(x)
#> [1] NA
```

It’s annoying when R is that cautious, but it is appropriate. You must think carefully about your situation. Does an `NA` in your data invalidate the statistic? If yes, then R is doing the right thing. If not, you can override this behavior by setting `na.rm=TRUE`, which tells R to ignore the `NA` values:

```
x <- c(0, 1, 1, 2, 3, NA)
sd(x, na.rm = TRUE)
#> [1] 1.14
```

In older versions of R, `mean` and `sd` were smart about data frames. They understood that each column of the data frame is a different variable, so they calculated their statistics for each column individually. This is no longer the case and, as a result, you may read confusing comments online or in older books (like the first edition of this book). In order to apply the functions to each column of a data frame we now need to use a helper function. The tidyverse family of helper functions for this sort of thing is in the `purrr` package. As with other tidyverse packages, this gets loaded when you run `library(tidyverse)`. The function we'll use to apply a function to each column of a data frame is `map_dbl`:

```
data(cars)

map_dbl(cars, mean)
#> speed dist
#> 15.4 43.0
map_dbl(cars, sd)
#> speed dist
#> 5.29 25.77
map_dbl(cars, median)
#> speed dist
#> 15 36
```

Notice in this example that `mean` and `sd` each return two values, one for each column defined by the data frame. (Technically, they return a two-element vector whose `names` attribute is taken from the columns of the data frame.)

The `var` function understands data frames without the help of a mapping function. It calculates the covariance between the columns of the data frame and returns the covariance matrix:

```
var(cars)
#>      speed dist
#> speed   28 110
#> dist    110 664
```

Likewise, if `x` is either a data frame or a matrix, then `cor(x)` returns the correlation matrix and `cov(x)` returns the covariance matrix:

```
cor(cars)
#>      speed dist
#> speed 1.000 0.807
#> dist   0.807 1.000
cov(cars)
#>      speed dist
#> speed   28 110
#> dist    110 664
```

See Also

See Recipe 2.14, Recipe 5.27, and Recipe 9.17.

2.7 Creating Sequences

Problem

You want to create a sequence of numbers.

Solution

Use an `n:m` expression to create the simple sequence $n, n+1, n+2, \dots, m$:

```
1:5  
#> [1] 1 2 3 4 5
```

Use the `seq` function for sequences with an increment other than 1:

```
seq(from = 1, to = 5, by = 2)  
#> [1] 1 3 5
```

Use the `rep` function to create a series of repeated values:

```
rep(1, times = 5)  
#> [1] 1 1 1 1 1
```

Discussion

The colon operator (`n:m`) creates a vector containing the sequence $n, n+1, n+2, \dots, m$:

```
0:9  
#> [1] 0 1 2 3 4 5 6 7 8 9  
10:19  
#> [1] 10 11 12 13 14 15 16 17 18 19  
9:0  
#> [1] 9 8 7 6 5 4 3 2 1 0
```

R was clever with the last expression (`9:0`). Because 9 is larger than 0, it counts backward from the starting to ending value. You can also use the colon operator directly with the pipe to pass data to another function:

```
10:20 %>% mean()
```

The colon operator works for sequences that grow by 1 only. The `seq` function also builds sequences but supports an optional third argument, which is the increment:

```
seq(from = 0, to = 20)  
#> [1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
seq(from = 0, to = 20, by = 2)  
#> [1] 0 2 4 6 8 10 12 14 16 18 20
```

```
seq(from = 0, to = 20, by = 5)
#> [1] 0 5 10 15 20
```

Alternatively, you can specify a length for the output sequence and then R will calculate the necessary increment:

```
seq(from = 0, to = 20, length.out = 5)
#> [1] 0 5 10 15 20
seq(from = 0, to = 100, length.out = 5)
#> [1] 0 25 50 75 100
```

The increment need not be an integer. R can create sequences with fractional increments, too:

```
seq(from = 1.0, to = 2.0, length.out = 5)
#> [1] 1.00 1.25 1.50 1.75 2.00
```

For the special case of a “sequence” that is simply a repeated value, you should use the `rep` function, which repeats its first argument:

```
rep(pi, times = 5)
#> [1] 3.14 3.14 3.14 3.14 3.14
```

See Also

See [Recipe 7.13](#) for creating a sequence of Date objects.

2.8 Comparing Vectors

Problem

You want to compare two vectors, or you want to compare an entire vector against a scalar.

Solution

The comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) can perform an element-by-element comparison of two vectors. They can also compare a vector’s element against a scalar. The result is a vector of logical values in which each value is the result of one element-wise comparison.

Discussion

R has two logical values, `TRUE` and `FALSE`. These are often called *Boolean* values in other programming languages.

The comparison operators compare two values and return `TRUE` or `FALSE`, depending upon the result of the comparison:

```

a <- 3
a == pi # Test for equality
#> [1] FALSE
a != pi # Test for inequality
#> [1] TRUE
a < pi
#> [1] TRUE
a > pi
#> [1] FALSE
a <= pi
#> [1] TRUE
a >= pi
#> [1] FALSE

```

You can experience the power of R by comparing entire vectors at once. R will perform an element-by-element comparison and return a vector of logical values, one for each comparison:

```

v <- c(3, pi, 4)
w <- c(pi, pi, pi)
v == w # Compare two 3-element vectors
#> [1] FALSE TRUE FALSE
v != w
#> [1] TRUE FALSE TRUE
v < w
#> [1] TRUE FALSE FALSE
v <= w
#> [1] TRUE TRUE FALSE
v > w
#> [1] FALSE FALSE TRUE
v >= w
#> [1] FALSE TRUE TRUE

```

You can also compare a vector against a single scalar, in which case R will expand the scalar to the vector's length and then perform the element-wise comparison. The previous example can be simplified in this way:

```

v <- c(3, pi, 4)
v == pi # Compare a 3-element vector against one number
#> [1] FALSE TRUE FALSE
v != pi
#> [1] TRUE FALSE TRUE

```

This is an application of the Recycling Rule discussed in [Recipe 5.3](#).

After comparing two vectors, you often want to know whether *any* of the comparisons were true or whether *all* the comparisons were true. The `any` and `all` functions handle those tests. They both test a logical vector. The `any` function returns `TRUE` if any element of the vector is `TRUE`. The `all` function returns `TRUE` if all elements of the vector are `TRUE`:

```
v <- c(3, pi, 4)
any(v == pi) # Return TRUE if any element of v equals pi
#> [1] TRUE
all(v == 0) # Return TRUE if all elements of v are zero
#> [1] FALSE
```

See Also

See Recipe 2.9.

2.9 Selecting Vector Elements

Problem

You want to extract one or more elements from a vector.

Solution

Select the indexing technique appropriate for your problem:

- Use square brackets to select vector elements by their position, such as `v[3]` for the third element of `v`.
- Use negative indexes to exclude elements.
- Use a vector of indexes to select multiple values.
- Use a logical vector to select elements based on a condition.
- Use names to access named elements.

Discussion

Selecting elements from vectors is another powerful feature of R. Basic selection is handled just as in many other programming languages—use square brackets and a simple index:

```
fib <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib
#> [1] 0 1 1 2 3 5 8 13 21 34
fib[1]
#> [1] 0
fib[2]
#> [1] 1
fib[3]
#> [1] 1
fib[4]
#> [1] 2
```

```
fib[5]
#> [1] 3
```

Notice that the first element has an index of 1, not 0 as in some other programming languages.

A cool feature of vector indexing is that you can select multiple elements at once. The index itself can be a vector, and each element of that indexing vector selects an element from the data vector:

```
fib[1:3] # Select elements 1 through 3
#> [1] 0 1 1
fib[4:9] # Select elements 4 through 9
#> [1] 2 3 5 8 13 21
```

An index of 1:3 means select elements 1, 2, and 3, as just shown. The indexing vector needn't be a simple sequence, however. You can select elements anywhere within the data vector—as in this example, which selects elements 1, 2, 4, and 8:

```
fib[c(1, 2, 4, 8)]
#> [1] 0 1 2 13
```

R interprets negative indexes to mean *exclude* a value. An index of -1, for instance, means exclude the first value and return all other values:

```
fib[-1] # Ignore first element
#> [1] 1 1 2 3 5 8 13 21 34
```

You can extend this method to exclude whole slices by using an indexing vector of negative indexes:

```
fib[1:3] # As before
#> [1] 0 1 1
fib[-(1:3)] # Invert sign of index to exclude instead of select
#> [1] 2 3 5 8 13 21 34
```

Another indexing technique uses a logical vector to select elements from the data vector. Everywhere that the logical vector is TRUE, an element is selected:

```
fib < 10 # This vector is TRUE wherever fib is less than 10
#> [1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE
fib[fib < 10] # Use that vector to select elements less than 10
#> [1] 0 1 1 2 3 5 8
fib %% 2 == 0 # This vector is TRUE wherever fib is even
#> [1] TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE
fib[fib %% 2 == 0] # Use that vector to select the even elements
#> [1] 0 2 8 34
```

Ordinarily, the logical vector should be the same length as the data vector so you are clearly either including or excluding each element. (If the lengths differ, then you need to understand the Recycling Rule, discussed in [Recipe 5.3](#).)

By combining vector comparisons, logical operators, and vector indexing, you can perform powerful selections with very little R code.

For example, you can select all elements greater than the median:

```
v <- c(3, 6, 1, 9, 11, 16, 0, 3, 1, 45, 2, 8, 9, 6, -4)
v[ v > median(v)]
#> [1] 9 11 16 45 8 9
```

or select all elements in the lower and upper 5%:

```
v[ (v < quantile(v, 0.05)) | (v > quantile(v, 0.95)) ]
#> [1] 45 -4
```

The previous example uses the `|` operator, which means “or” when indexing. If you wanted “and,” you would use the `&` operator.

You can also select all elements that exceed ± 1 standard deviations from the mean:

```
v[ abs(v - mean(v)) > sd(v)]
#> [1] 45 -4
```

or select all elements that are neither NA nor NULL:

```
v <- c(1, 2, 3, NA, 5)
v[!is.na(v) & !is.null(v)]
#> [1] 1 2 3 5
```

One final indexing feature lets you select elements by name. It assumes that the vector has a `names` attribute, defining a name for each element. You can define the names by assigning a vector of character strings to the attribute:

```
years <- c(1960, 1964, 1976, 1994)
names(years) <- c("Kennedy", "Johnson", "Carter", "Clinton")
years
#> Kennedy Johnson Carter Clinton
#> 1960      1964      1976      1994
```

Once the names are defined, you can refer to individual elements by name:

```
years["Carter"]
#> Carter
#> 1976
years["Clinton"]
#> Clinton
#> 1994
```

This generalizes to allow indexing by vectors of names; R returns every element named in the index:

```
years[c("Carter", "Clinton")]
#> Carter Clinton
#> 1976      1994
```

See Also

See [Recipe 5.3](#) for more about the Recycling Rule.

2.10 Performing Vector Arithmetic

Problem

You want to operate on an entire vector at once.

Solution

The usual arithmetic operators can perform element-wise operations on entire vectors. Many functions operate on entire vectors, too, and return a vector result.

Discussion

Vector operations are one of R's great strengths. All the basic arithmetic operators can be applied to pairs of vectors. They operate in an element-wise manner; that is, the operator is applied to corresponding elements from both vectors:

```
v <- c(11, 12, 13, 14, 15)
w <- c(1, 2, 3, 4, 5)
v + w
#> [1] 12 14 16 18 20
v - w
#> [1] 10 10 10 10 10
v * w
#> [1] 11 24 39 56 75
v / w
#> [1] 11.00 6.00 4.33 3.50 3.00
w^v
#> [1] 1.00e+00 4.10e+03 1.59e+06 2.68e+08 3.05e+10
```

Observe that the length of the result here is equal to the length of the original vectors. The reason is that each element comes from a pair of corresponding values in the input vectors.

If one operand is a vector and the other is a scalar, then the operation is performed between every vector element and the scalar:

```
w
#> [1] 1 2 3 4 5
w + 2
#> [1] 3 4 5 6 7
w - 2
#> [1] -1 0 1 2 3
w * 2
#> [1] 2 4 6 8 10
```

```
w / 2  
#> [1] 0.5 1.0 1.5 2.0 2.5  
2^w  
#> [1] 2 4 8 16 32
```

For example, you can recenter an entire vector in one expression simply by subtracting the mean of its contents:

```
w  
#> [1] 1 2 3 4 5  
mean(w)  
#> [1] 3  
w - mean(w)  
#> [1] -2 -1 0 1 2
```

Likewise, you can calculate the z -score of a vector in one expression—subtract the mean and divide by the standard deviation:

```
w  
#> [1] 1 2 3 4 5  
sd(w)  
#> [1] 1.58  
(w - mean(w)) / sd(w)  
#> [1] -1.265 -0.632 0.000 0.632 1.265
```

Yet the implementation of vector-level operations goes far beyond elementary arithmetic. It pervades the language, and many functions operate on entire vectors. The functions `sqrt` and `log`, for example, apply themselves to every element of a vector and return a vector of results:

```
w <- 1:5  
w  
#> [1] 1 2 3 4 5  
sqrt(w)  
#> [1] 1.00 1.41 1.73 2.00 2.24  
log(w)  
#> [1] 0.000 0.693 1.099 1.386 1.609  
sin(w)  
#> [1] 0.841 0.909 0.141 -0.757 -0.959
```

There are two great advantages to vector operations. The first and most obvious is convenience. Operations that require looping in other languages are one-liners in R. The second is speed. Most vectorized operations are implemented directly in C code, so they are substantially faster than the equivalent R code you could write.

See Also

Performing an operation between a vector and a scalar is actually a special case of the Recycling Rule; see [Recipe 5.3](#).

2.11 Getting Operator Precedence Right

Problem

Your R expression is producing a curious result, and you wonder if operator precedence is causing problems.

Solution

The full list of operators is shown in [Table 2-1](#), listed in order of precedence from highest to lowest. Operators of equal precedence are evaluated from left to right except where indicated.

Table 2-1. Operator precedence

Operator	Meaning	See also
[[Indexing	Recipe 2.9
:: :::	Access variables in a namespace (environment)	
\$ @	Component extraction, slot extraction	
^	Exponentiation (right to left)	
- +	Unary minus and plus	
:	Sequence creation	Recipe 2.7 , Recipe 7.13
%any% (including %>%)	Special operators	Discussion (this recipe)
* /	Multiplication, division	Discussion (this recipe)
+ -	Addition, subtraction	
== != < > <= >=	Comparison	Recipe 2.8
!	Logical negation	
& &&	Logical “and,” short-circuit “and”	
	Logical “or,” short-circuit “or”	
~	Formula	Recipe 11.1
-> ->>	Rightward assignment	Recipe 2.2
=	Assignment (right to left)	Recipe 2.2
<- <<-	Assignment (right to left)	Recipe 2.2
?	Help	Recipe 1.8

It's not important that you know what every one of these operators does, or what they mean. The list here is intended simply to expose you to the idea that different operators have different precedence.

Discussion

Getting your operator precedence wrong in R is a common problem. It certainly happens to us a lot. We unthinkingly expect that the expression `0:n-1` will create a sequence of integers from 0 to $n-1$, but it does not:

```
n <- 10  
0:n - 1  
#> [1] -1 0 1 2 3 4 5 6 7 8 9
```

It creates the sequence from -1 to $n-1$ because R interprets it as $(0:n)-1$.

You might not recognize the notation `%any%` in the table. R interprets any text between percent signs (`%...%`) as a binary operator. Several such operators have predefined meanings:

<code>%%</code>	Modulo operator
<code>%/%</code>	Integer division
<code>%*%</code>	Matrix multiplication
<code>%in%</code>	Returns TRUE if the left operand occurs in its right operand; FALSE otherwise
<code>%>%</code>	Pipe that passes results from the left to a function on the right

You can also define new binary operators using the `%...%` notation; see [Recipe 12.17](#). The point here is that all such operators have the same precedence.

See Also

See [Recipe 2.10](#) for more about vector operations, [Recipe 5.15](#) for more about matrix operations, and [Recipe 12.17](#) to define your own operators. See also the Arithmetic and Syntax topics in the R help pages as well as Chapters 5 and 6 of [R in a Nutshell](#).

2.12 Typing Less and Accomplishing More

Problem

You are getting tired of typing long sequences of commands, and especially tired of typing the same ones over and over.

Solution

Open an editor window and accumulate your reusable blocks of R commands there. Then, execute those blocks directly from that window. Reserve the console window for typing brief or one-off commands.

When you are done, you can save the accumulated code blocks in a script file for later use.

Discussion

The typical R beginner types an expression in the console window and sees what happens. As he gets more comfortable, he types increasingly complicated expressions. Then he begins typing multiline expressions. Soon, he is typing the same multiline expressions over and over, perhaps with small variations, in order to perform his increasingly complicated calculations.

The experienced R user does not often retype a complex expression. She may type the same expression once or twice, but when she realizes it is useful and reusable she will cut and paste it into an editor window. To execute the snippet thereafter, she selects the snippet in the editor window and tells R to execute it, rather than retyping it. This technique is especially powerful as her snippets evolve into long blocks of code.

In RStudio, a few shortcuts in the IDE facilitate this work style. Windows and Linux machines have slightly different keys than Mac machines: Windows/Linux uses the *Ctrl* and *Alt* modifiers, whereas the Mac uses *Cmd* and *Opt*.

To open an editor window

From the main menu, select File → New File, then select the type of file you want to create—in this case, an R script. Or if you know you want an R script, you can press Shift-Ctrl-N (Windows) or Shift-Cmd-N (Mac).

To execute one line of the editor window

Position the cursor on the line and then press Ctrl-Enter (Windows) or Cmd-Enter (Mac) to execute it.

To execute several lines of the editor window

Highlight the lines using your mouse; then press Ctrl-Enter (Windows) or Cmd-Enter (Mac) to execute them.

To execute the entire contents of the editor window

Press Ctrl-Alt-R (Windows) or Cmd-Opt-R (Mac) to execute the whole editor window. Or from the menu, click Code → Run Region → Run All.

You can find these keyboard shortcuts and dozens more within RStudio by choosing the Tools → Keyboard Shortcuts Help menu item.

Reproducing lines from the console window in the editor window is simply a matter of copy and paste. When you exit RStudio, it will ask if you want to save the new script. You can either save it for future reuse or discard it.

2.13 Creating a Pipeline of Function Calls

Problem

Creating many intermediate variables in your code is tedious and overly verbose, while nesting R functions makes the code nearly unreadable.

Solution

Use the pipe operator (`%>%`) to make your expressions easier to read and write. The pipe operator, created by Stefan Bache and found in the `magrittr` package, is used extensively in many `tidyverse` functions as well.

Use the pipe operator to combine multiple functions together into a “pipeline” of functions without intermediate variables:

```
library(tidyverse)
data(mpg)

mpg %>%
  filter(cty > 21) %>%
  head(3) %>%
  print()

#> # A tibble: 3 x 11
#>   manufacturer model  displ year cyl trans drv    cty    hwy fl     class
#>   <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
#> 1 chevrolet    malibu  2.4  2008     4 auto~ f      22     30 r     mids~
#> 2 honda         civic   1.6  1999     4 manu~ f      28     33 r     subc~
#> 3 honda         civic   1.6  1999     4 auto~ f      24     32 r     subc~
```

Using the pipe is much cleaner and easier to read than using intermediate temporary variables:

```
temp1 <- filter(mpg, cty > 21)
temp2 <- head(temp1, 3)
print(temp2)

#> # A tibble: 3 x 11
#>   manufacturer model  displ year cyl trans drv    cty    hwy fl     class
#>   <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
#> 1 chevrolet    malibu  2.4  2008     4 auto~ f      22     30 r     mids~
#> 2 honda         civic   1.6  1999     4 manu~ f      28     33 r     subc~
#> 3 honda         civic   1.6  1999     4 auto~ f      24     32 r     subc~
```

Discussion

The pipe operator does not provide any new functionality to R, but it can greatly improve the readability of code. It takes the output of the function or object on the left of the operator and passes it as the first argument of the function on the right.

Writing this:

```
x %>% head()
```

is functionally the same as writing this:

```
head(x)
```

In both cases `x` is the argument to `head`. We can supply additional arguments, but `x` is always the *first* argument. These two lines are also functionally identical:

```
x %>% head(n = 10)
```

```
head(x, n = 10)
```

This difference may seem small, but with a more complicated example, the benefits begin to accumulate. If we had a workflow where we wanted to use `filter` to limit our data to values, then `select` to keep only certain variables, followed by `ggplot` to create a simple plot, we could use intermediate variables:

```
library(tidyverse)

filtered_mpg <- filter(mpg, cty > 21)
selected_mpg <- select(filtered_mpg, cty, hwy)
ggplot(selected_mpg, aes(cty, hwy)) + geom_point()
```

This incremental approach is fairly readable but creates a number of intermediate data frames and requires the user to keep track of the state of many objects, which can add cognitive load. But the code does produce the desired graph.

An alternative is to nest the functions together:

```
ggplot(select(filter(mpg, cty > 21), cty, hwy), aes(cty, hwy)) + geom_point()
```

While this is very concise since it's only one line, this code requires much more attention to read and understand what's going on. Code that is difficult for the user to parse mentally can introduce potential for error, and can also be harder to maintain in the future. Instead, we can use pipes:

```
mpg %>%
  filter(cty > 21) %>%
  select(cty, hwy) %>%
  ggplot(aes(cty, hwy)) + geom_point()
```

The preceding code starts with the `mpg` dataset and pipes it to the `filter` function, which keeps only records where the city mpg value (`cty`) is greater than 21. Those results are piped into the `select` command, which keeps only the listed variables `cty`

and `hwy`, and in turn those are piped into the `ggplot` command, which produces the point plot in [Figure 2-2](#).

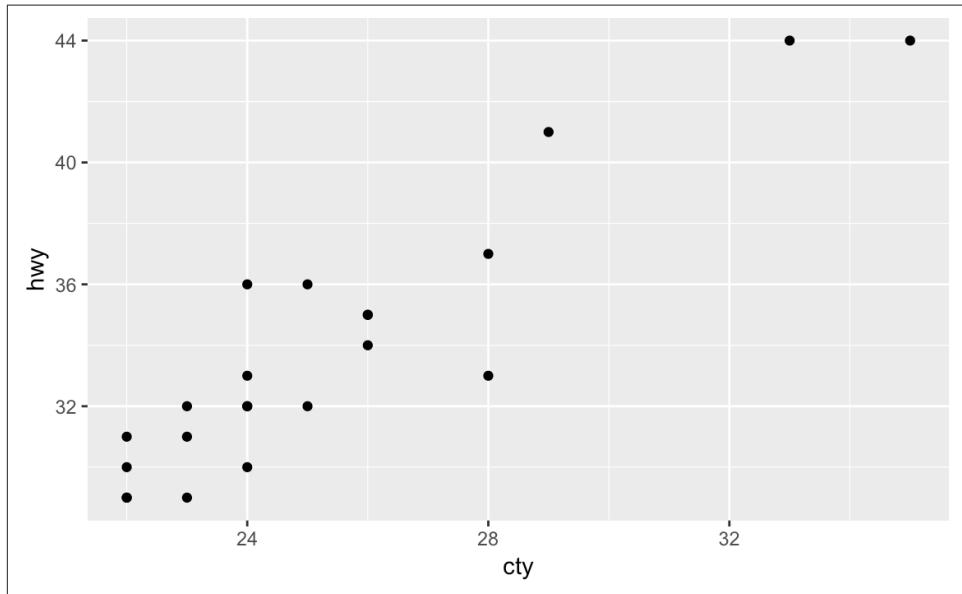


Figure 2-2. Plotting with pipes example

If you want the argument going into your target (righthand side) function to be somewhere other than the first argument, use the dot (.) operator. So this:

```
iris %>% head(3)
```

is the same as:

```
iris %>% head(3, x = .)
```

However, in the second example we passed the `iris` data frame into the second named argument using the dot operator. This can be handy for functions where the input data frame goes in a position other than the first argument.

Throughout this book we use pipes to hold together data transformations with multiple steps. We typically format the code with a line break after each pipe and then indent the code on the following lines. This makes the code easily identifiable as parts of the same data pipeline.

2.14 Avoiding Some Common Mistakes

Problem

You want to avoid some of the common mistakes made by beginning users—and by experienced users, for that matter!

Discussion

Here are some easy ways to make trouble for yourself.

Forgetting the parentheses after a function invocation

You call an R function by putting parentheses after the name. For instance, this line invokes the `ls` function:

```
ls()
```

However, if you omit the parentheses, R does not execute the function. Instead, it shows the function definition, which is almost never what you want:

```
ls
```

```
#> function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
#>           pattern, sorted = TRUE)
#> {
#>   if (!missing(name)) {
#>     pos <- tryCatch(name, error = function(e) e)
#>     if (inherits(pos, "error")) {
#>       name <- substitute(name)
#>       if (!is.character(name))
#>         name <- deparse(name)
#>     # etc.
```

Mistyping “`<-`” as “`<(space)-`”

The assignment operator is `<-`, with no space between the `<` and the `-`:

```
x <- pi # Set x to 3.141592...
```

If you accidentally insert a space between `<` and `-`, the meaning changes completely:

```
x < -pi # Oops! We are comparing x instead of setting it!
#> [1] FALSE
```

This is now a comparison (`<`) between `x` and `-pi` (negative π). It does not change `x`. If you are lucky, `x` is undefined and R will complain, alerting you that something is fishy:

```
x < -pi
#> Error in eval(expr, envir, enclos): object 'x' not found
```

If `x` is defined, R will perform the comparison and print a logical value, `TRUE` or `FALSE`. That should alert you that something is wrong, as an assignment does not normally print anything:

```
x <- 0 # Initialize x to zero
x < -pi # Oops!
#> [1] FALSE
```

Incorrectly continuing an expression across lines

R reads your typing until you finish a complete expression, no matter how many lines of input that requires. It prompts you for additional input using the `+` prompt until it is satisfied. This example splits an expression across two lines:

```
total <- 1 + 2 + 3 + # Continued on the next line
      4 +
print(total)
#> [1] 15
```

Problems begin when you accidentally finish the expression prematurely, which can easily happen:

```
total <- 1 + 2 + 3 # Oops! R sees a complete expression
+ 4 + 5 # This is a new expression; R prints its value
#> [1] 9
print(total)
#> [1] 6
```

There are two clues that something is amiss: R prompted you with a normal prompt (`>`), not the continuation prompt (`+`), and it printed the value of $4 + 5$.

This common mistake is a headache for the casual user. It is a nightmare for programmers, however, because it can introduce hard-to-find bugs into R scripts.

Using `=` instead of `==`

Use the double-equals operator (`==`) for comparisons. If you accidentally use the single-equals operator (`=`), you will irreversibly overwrite your variable:

```
v <- 1 # Assign 1 to v
v == 0 # Compare v against zero
#> [1] FALSE
v = 0 # Assign 0 to v, overwriting previous contents
print(v)
#> [1] 0
```

Writing `1:n+1` when you mean `1:(n+1)`

You might think that `1:n+1` is the sequence of numbers $1, 2, \dots, n, n+1$. It's not. It is the sequence $1, 2, \dots, n$ with 1 added to every element, giving $2, 3, \dots, n, n+1$. This

happens because R interprets `1:n+1` as `(1:n)+1`. Use parentheses to get exactly what you want:

```
n <- 5
1:n + 1
#> [1] 2 3 4 5 6
1:(n + 1)
#> [1] 1 2 3 4 5 6
```

Getting bitten by the Recycling Rule

Vector arithmetic and vector comparisons work well when both vectors have the same length. However, the results can be baffling when the operands are vectors of differing lengths. Guard against this possibility by understanding and remembering the Recycling Rule (see [Recipe 5.3](#)).

Installing a package but not loading it with library or require

Installing a package is the first step toward using it, but one more step is required. Use `library` or `require` to load the package into your search path. Until you do so, R will not recognize the functions or datasets in the package (see [Recipe 3.8](#)):

```
x <- rnorm(100)
n <- 5
truehist(x, n)
#> Error in truehist(x, n): could not find function "truehist"
```

However, if you load the library first, then the code runs and you get the chart shown in [Figure 2-3](#):

```
library(MASS) # Load the MASS package into R
truehist(x, n)
```

We typically use `library` instead of `require`. The reason is that if you create an R script that uses `library` and the desired package is not already installed, R will return an error. In contrast, `require` will simply return `FALSE` if the package is not installed.

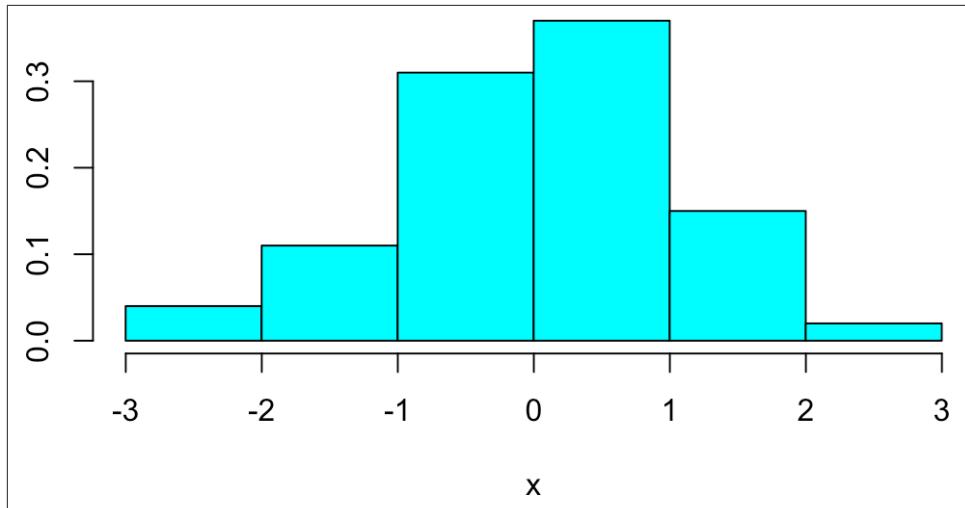


Figure 2-3. Example truehist

Writing `lst[n]` when you mean `lst[[n]]` or vice versa

If the variable `lst` contains a list, it can be indexed in two ways: `lst[[n]]` is the n th element of the list, whereas `lst[n]` is a list whose only element is the n th element of `lst`. That's a big difference. See [Recipe 5.7](#).

Using & instead of &&, or vice versa; same for | and ||

Use `&` and `|` in logical expressions involving the logical values `TRUE` and `FALSE`. See [Recipe 2.9](#).

Use `&&` and `||` for the flow-of-control expressions inside `if` and `while` statements.

Programmers accustomed to other programming languages may reflexively use `&&` and `||` everywhere because “they are faster.” But those operators give peculiar results when applied to vectors of logical values, so avoid them unless you are sure that they do what you want.

Passing multiple arguments to a single-argument function

What do you think is the value of `mean(9,10,11)`? No, it’s not 10. It’s 9. The `mean` function computes the mean of the first argument. The second and third arguments are being interpreted as other positional arguments. To pass multiple items into a single argument, we put them in a vector with the `c` operator. `mean(c(9,10,11))` will return 10, as you might expect.

Some functions, such as `mean`, take one argument. Other arguments, such as `max` and `min`, take multiple arguments and apply themselves across all arguments. Be sure you know which are which.

Thinking that `max` behaves like `pmax`, or that `min` behaves like `pmin`

The `max` and `min` functions have multiple arguments and return one value: the maximum or minimum of all their arguments.

The `pmax` and `pmin` functions have multiple arguments but return a vector with values taken element-wise from the arguments. For more info, see [Recipe 12.8](#).

Misusing a function that does not understand data frames

Some functions are quite clever regarding data frames. They apply themselves to the individual columns of the data frame, computing their result for each individual column. Sadly, not all functions are that bright. This includes the `mean`, `median`, `max`, and `min` functions. They will lump together every value from every column and compute their result from the lump, or possibly just return an error. Be aware of which functions are savvy to data frames and which are not. When in doubt, read the documentation for the function you are considering.

Using a single backslash (\) in Windows paths

It's common to copy and paste filepaths into your R scripts, but if you're using R on Windows you need to take care. Windows File Explorer may show you that your path is `C:\temp\my_file.csv`, but if you try to tell R to read that file, you'll get a cryptic message:

```
Error: '\m' is an unrecognized escape in character string starting "'.\temp\m"
```

This is because R sees backslashes as special characters. You can get around this by using either forward slashes (/) or double backslashes (\\\):

```
read_csv(`./temp/my_file.csv`)
read_csv(`.\temp\\my_file.csv`)
```

This is only an issue on Windows because both Mac and Linux use forward slashes as path separators.

Posting a question to Stack Overflow or the mailing list before searching for the answer

Don't waste your time. Don't waste other people's time. Before you post a question to a mailing list or to Stack Overflow, do your homework and search the archives. Odds are, someone has already answered your question. If so, you'll see the answer in the discussion thread for the question. See [Recipe 1.13](#).

See Also

See Recipes [1.13](#), [2.9](#), [3.8](#), [5.3](#), [5.7](#), and [12.8](#).

Navigating the Software

Both R and RStudio are big chunks of software, first and foremost. You will inevitably spend time doing what one does with any big piece of software: configuring it, customizing it, updating it, and fitting it into your computing environment. This chapter will help you perform those tasks. There is nothing here about numerics, statistics, or graphics. This is all about dealing with R and RStudio as software.

3.1 Getting and Setting the Working Directory

Problem

You want to change your working directory, or you just want to know what it is.

Solution

RStudio

Navigate to a directory in the Files pane. Then from the Files pane, select More → Set As Working Directory, as shown in [Figure 3-1](#).

Console

Use `getwd` to report the working directory, and use `setwd` to change it:

```
getwd()  
#> [1] "/Volumes/SecondDrive/jal/DocumentsPersonal/R-Cookbook"  
setwd("~/Documents/MyDirectory")
```

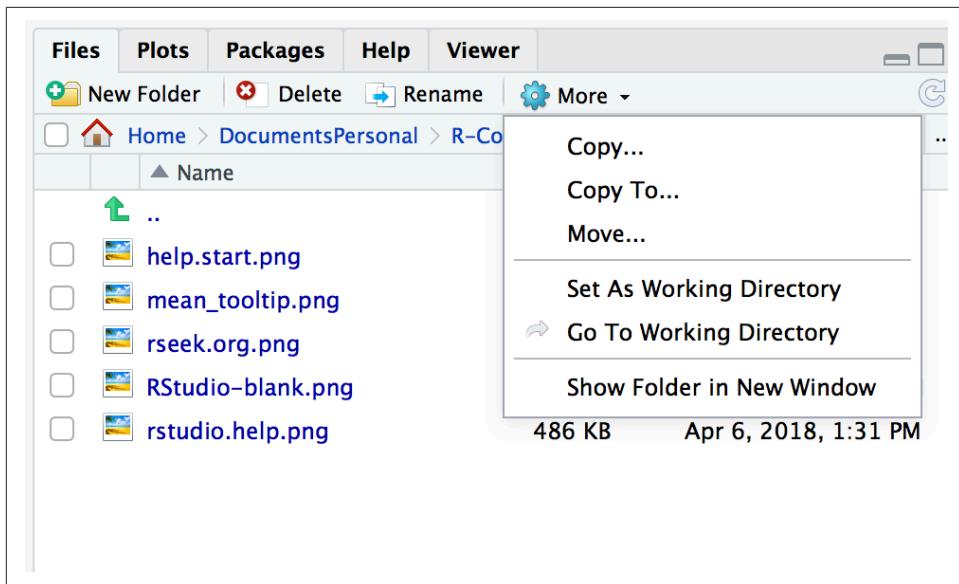


Figure 3-1. RStudio: Set As Working Directory

Discussion

Your working directory is important because it is the default location for all file input and output—including reading and writing data files, opening and saving script files, and saving your workspace image. When you open a file and do not specify an absolute path, R will assume that the file is in your working directory.

If you’re using RStudio projects, your default working directory will be the home directory of the project. See [Recipe 3.2](#) for more about creating RStudio projects.

See Also

See [Recipe 4.5](#) for dealing with filenames in Windows.

3.2 Creating a New RStudio Project

Problem

You want to create a new RStudio project to keep all your files related to a specific project.

Solution

Click File → New Project as in [Figure 3-2](#).

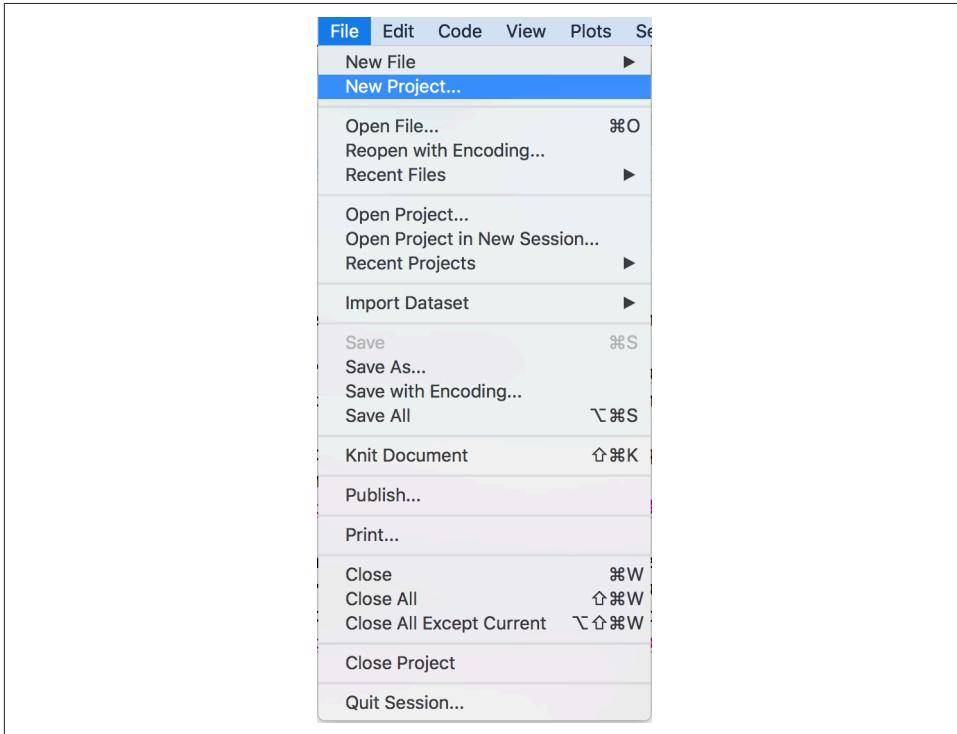


Figure 3-2. Creating a new project

This will open the New Project dialog box and allow you to choose which type of project you would like to create, as shown in [Figure 3-3](#).

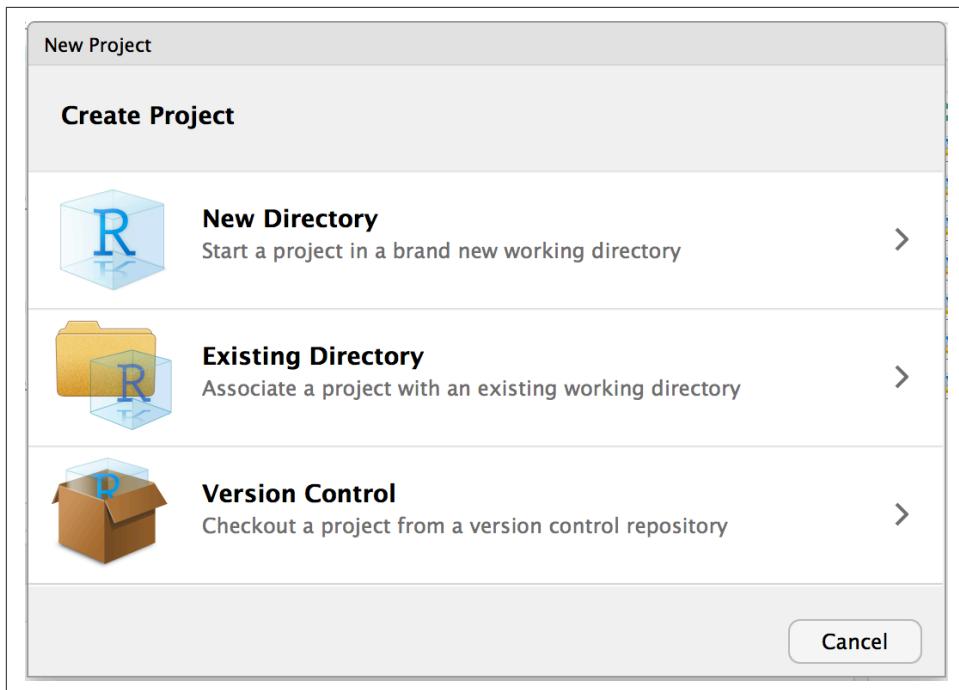


Figure 3-3. New Project dialog

Discussion

Projects are a powerful concept that's specific to RStudio. They help you by doing the following:

- Setting your working directory to the project directory.
- Preserving window state in RStudio so when you return to a project your windows are all as you left them. This includes opening any files you had open when you last saved your project.
- Preserving RStudio project settings.

To hold your project settings, RStudio creates a project file with an *.Rproj* extension in the project directory. If you open the project file in RStudio, it works like a shortcut for opening the project. In addition, RStudio creates a hidden directory named *.Rproj.user* to house temporary files related to your project.

Any time you're working on something nontrivial in R we recommend creating an RStudio project. Projects help you stay organized and make your project workflow easier.

3.3 Saving Your Workspace

Problem

You want to save your workspace and all variables and functions you have in memory.

Solution

Call the `save.image()` function:

```
save.image()
```

Discussion

Your workspace holds your R variables and functions, and it is created when R starts. The workspace is held in your computer's main memory and lasts until you exit from R. You can easily view the contents of your workspace in RStudio in the Environment tab, as shown in [Figure 3-4](#).

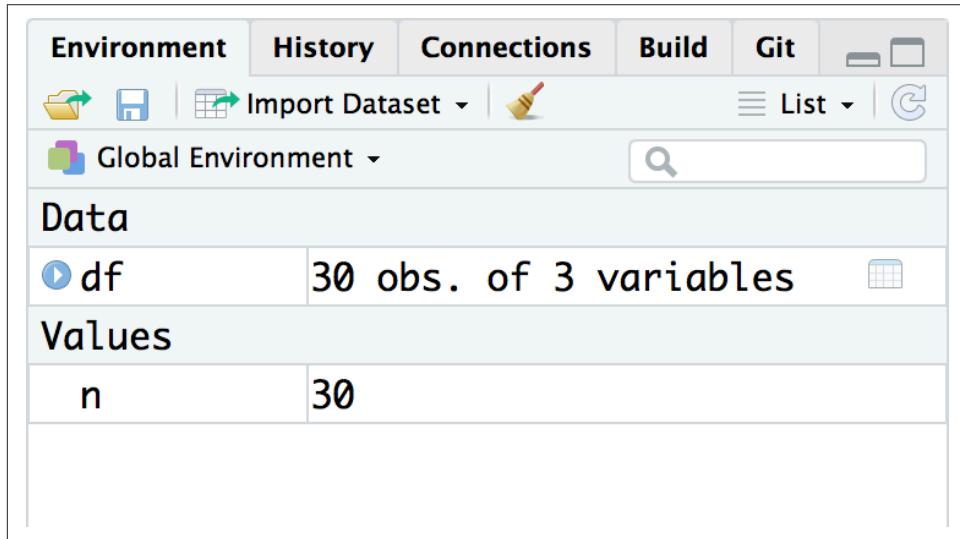


Figure 3-4. RStudio Environment pane

However, you may want to save your workspace without exiting R, because you know bad things mysteriously happen when you close your laptop to carry it home. In this case, use the `save.image` function.

The workspace is written to a file called `.RData` in the working directory. When R starts, it looks for that file and, if it finds it, initializes the workspace from it.

Sadly, the workspace does not include your open graphs: for example, that cool graph on your screen disappears when you exit R. The workspace also does not include the positions of your windows or your RStudio settings. This is why we recommend using RStudio projects and writing your R scripts so that you can reproduce everything you've created.

See Also

See [Recipe 3.1](#) for setting the working directory.

3.4 Viewing Your Command History

Problem

You want to see your recent sequence of commands.

Solution

Depending on what you are trying to accomplish, you can use a few different methods to access your prior command history. If you are in the RStudio console pane, you can press the up arrow to interactively scroll through past commands.

If you want to see a listing of past commands, you can either execute the `history()` function or access the History pane in RStudio to view your most recent input:

```
history()
```

In RStudio typing `history()` into the console simply activates the History pane ([Figure 3-5](#)). You could also make that pane visible by clicking on it with your cursor.

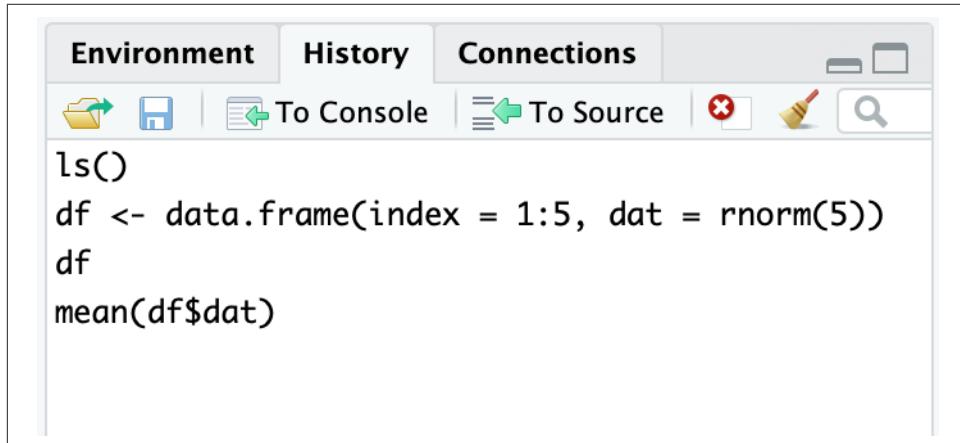


Figure 3-5. RStudio History pane

Discussion

The `history` function displays your most recent commands. In RStudio the `history` command will activate the History pane. If you're running R outside of RStudio, `history` shows the most recent 25 lines, but you can request more like so:

```
history(100)      # Show 100 most recent lines of history  
history(Inf)      # Show entire saved history
```

From within RStudio, the History tab shows an exhaustive list of past commands in chronological order, with the most recent at the bottom of the list. You can highlight past commands with your cursor, then click on “To Console” or “To Source” to copy past commands into the console or source editor, respectively. This can be terribly handy when you've done interactive data analysis and then decide you want to save some past steps to a source file for later use.

From the console you can see your history by simply pressing the up arrow to scroll backward through your input, which causes your previous typing to reappear, one line at a time.

If you've exited from R or RStudio, you can still see your command history. R saves the history in a file called `.Rhistory` in the working directory. Open the file with a text editor and then scroll to the bottom; you will see your most recent typing.

3.5 Saving the Result of the Previous Command

Problem

You typed an expression into R that calculated a value, but you forgot to save the result in a variable.

Solution

A special variable called `.Last.value` saves the value of the most recently evaluated expression. Save it to a variable before you type anything else.

Discussion

It is frustrating to type a long expression or call a long-running function but then forget to save the result. Fortunately, you needn't retype the expression nor invoke the function again—the result was saved in the `.Last.value` variable:

```
aVeryLongRunningFunction() # Oops! Forgot to save the result!  
x <- .Last.value          # Capture the result now
```

A word of caution here: the contents of `.Last.value` are overwritten every time you type another expression, so capture the value immediately. If you don't remember until another expression has been evaluated, it's too late!

See Also

See [Recipe 3.4](#) to recall your command history.

3.6 Displaying Loaded Packages via the Search Path

Problem

You want to see the list of packages currently loaded into R.

Solution

Use the `search` function with no arguments:

```
search()
```

Discussion

The search path is a list of packages that are currently loaded into memory and available for use. Although many packages may be installed on your computer, only a few of them are actually loaded into the R interpreter at any given moment. You might be wondering which packages are loaded right now.

With no arguments, the `search` function returns the list of loaded packages. It produces output like this:

```
search()
#> [1] ".GlobalEnv"      "package:knitr"     "package:forcats"
#> [4] "package:stringr" "package:dplyr"    "package:purrr"
#> [7] "package:readr"    "package:tidyverse" "package:tibble"
#> [10] "package:ggplot2"  "package:tidyverse" "package:stats"
#> [13] "package:graphics" "package:grDevices" "package:utils"
#> [16] "package:datasets" "package:methods"  "Autoloads"
#> [19] "package:base"
```

Your machine may return a different result, depending on what's installed there. The return value of `search` is a vector of strings. The first string is `".GlobalEnv"`, which refers to your workspace. Most strings have the form `"package:packagename"`, which indicates that the package called `packagename` is currently loaded into R. In the preceding example, you can see many tidyverse packages installed, including `purrr`, `ggplot2`, and `tibble`.

R uses the search path to find functions. When you type a function name, R searches the path—in the order shown—until it finds the function in a loaded package. If the function is found, R executes it. Otherwise, it prints an error message and stops. (There is actually a bit more to it: the search path can contain environments, not just packages, and the search algorithm is different when initiated by an object within a package; see the [R Language Definition](#) for details.)

Since your workspace (`.GlobalEnv`) is first in the list, R looks for functions in your workspace before searching any packages. If your workspace and a package both contain a function with the same name, your workspace will “mask” the function; this means that R stops searching after it finds your function and so never sees the package function. This is a blessing if you want to override the package function...and a curse if you still want access to it. If you find yourself feeling cursed because you (or some package you loaded) overrode a function (or other object) from an existing loaded package, you can use the full `environment::name` form to call an object from a loaded package environment. For example, if you wanted to call the `dplyr` function `count`, you could do so using `dplyr::count`. Using the full explicit name to call a function will work even if you have not loaded the package, so if you have `dplyr` installed but not loaded, you can still call `dplyr::count`.



It is becoming increasingly common with online examples to show the full `packagename::function` in examples. While this removes ambiguity about where a function comes from, it makes example code very wordy.

Note that R will include only *loaded* packages in the search path. So if you have installed a package but not loaded it by using `library(packagename)`, then R will not add that package to the search path.

R also uses the search path to find R datasets (not files) or any other object via a similar procedure.

Unix and Mac users: don’t confuse the R search path with the Unix search path (the `PATH` environment variable). They are conceptually similar but two distinct things. The R search path is internal to R and is used by R only to locate functions and datasets, whereas the Unix search path is used by the OS to locate executable programs.

See Also

See [Recipe 3.8](#) for loading packages into R and [Recipe 3.7](#) for viewing the list of installed packages (not just loaded packages).

3.7 Viewing the List of Installed Packages

Problem

You want to know what packages are installed on your machine.

Solution

Use the `library` function with no arguments for a basic list. Use `installed.packages` to see more detailed information about the packages.

Discussion

The `library` function with no arguments prints a list of installed packages:

```
library()
```

The list can be quite long. In RStudio, it is displayed in a new tab in the editor window.

You can get more details via the `installed.packages` function, which returns a matrix of information regarding the packages on your machine. Each row corresponds to one installed package. The columns contain information such as the package name, library path, and version. The information is taken from R's internal database of installed packages.

To extract useful information from this matrix, use normal indexing methods. The following snippet calls `installed.packages` and extracts both the `Package` and `Version` columns for the first five packages, letting you see what version of each package is installed:

```
installed.packages()[1:5, c("Package", "Version")]
#>          Package    Version
#> abind      "abind"   "1.4-5"
#> ade4       "ade4"    "1.7-13"
#> adegenet   "adegenet" "2.1.1"
#> analogsea  "analogsea" "0.6.6.9110"
#> ape         "ape"     "5.3"
```

See Also

See [Recipe 3.8](#) for loading a package into memory.

3.8 Accessing the Functions in a Package

Problem

A package installed on your computer is either a standard package or a package you've downloaded. When you try using functions in the package, however, R cannot find them.

Solution

Use either the `library` function or the `require` function to load the package into R:

```
library(packagename)
```

Discussion

R comes with several standard packages, but not all of them are automatically loaded when you start R. Likewise, you can download and install many useful packages from CRAN or GitHub, but they are not automatically loaded when you run R. The MASS package comes standard with R, for example, but you could get this message when using the `lda` function in that package:

```
lda(x)
#> Error in lda(x): could not find function "lda"
```

R is complaining that it cannot find the `lda` function among the packages currently loaded into memory.

When you use the `library` function or the `require` function, R loads the package into memory and its contents become immediately available to you:

```
my_model <-
  lda(cty ~ displ + year, data = mpg)
#> Error in lda(cty ~ displ + year, data = mpg): could not find function "lda"

library(MASS)                                # Load the MASS library into memory
#>
#> Attaching package: 'MASS'
#> The following object is masked from 'package:dplyr':
#>
#>   select
my_model <-
  lda(cty ~ displ + year, data = mpg) # Now R can find the function
```

Before you call `library`, R does not recognize the function name. Afterward, the package contents are available and calling the `lda` function works.

Notice that you needn't enclose the package name in quotes.

The `require` function is nearly identical to `library`, but it has two features that are useful for writing scripts. It returns `TRUE` if the package was successfully loaded and `FALSE` otherwise. It also generates a mere warning if the load fails—unlike `library`, which generates an error.

Both functions have a key feature: they do not reload packages that are already loaded, so calling twice for the same package is harmless. This is especially nice for writing scripts. The script can load needed packages while knowing that loaded packages will not be reloaded.

The `detach` function will unload a package that is currently loaded:

```
detach(package:MASS)
```

Observe that the package name must be qualified, as in `package:MASS`.

One reason to unload a package is that it contains a function whose name conflicts with a same-named function lower on the search list. When such a conflict occurs, we say the higher function *masks* the lower function. You no longer “see” the lower function because R stops searching when it finds the higher function. Hence, unloading the higher package unmasks the lower name.

See Also

See [Recipe 3.6](#).

3.9 Accessing Built-in Datasets

Problem

You want to use one of R’s built-in datasets, or you want to access one of the datasets that comes with another package.

Solution

The standard datasets distributed with R are already available to you, since the `datasets` package is in your search path. If you’ve loaded any other packages, datasets that come with those loaded packages will also be available in your search path.

To access datasets in other packages, use the `data` function while giving the dataset name and package name:

```
data(dsname, package = "pkgname")
```

Discussion

R comes with many built-in datasets. Other packages, such as `dplyr` and `ggplot2`, also come with example data that's used in the examples found in their help files. These datasets are useful when you are learning about R, since they provide data with which to experiment.

Many datasets are kept in a package called (naturally enough) `datasets`, which is distributed with R. That package is in your search path, so you have instant access to its contents. For example, you can use the built-in dataset called `pressure`:

```
head(pressure)
#>   temperature pressure
#> 1          0  0.0002
#> 2         20  0.0012
#> 3         40  0.0060
#> 4         60  0.0300
#> 5         80  0.0900
#> 6        100  0.2700
```

If you want to know more about `pressure`, use the `help` function to learn about it:

```
help(pressure)      # Bring up help page for pressure dataset
```

You can see a table of contents for `datasets` by calling the `data` function with no arguments:

```
data()              # Bring up a list of datasets
```

Any R package can elect to include datasets that supplement those supplied in `datasets`. The `MASS` package, for example, includes many interesting datasets. Use the `data` function with the `package` argument to load a dataset from a specific package. `MASS` includes a dataset called `Cars93`, which you can load into memory in this way:

```
data(Cars93, package = "MASS")
```

After this call to `data`, the `Cars93` dataset is available to you; then you can execute `summary(Cars93)`, `head(Cars93)`, and so forth.

When attaching a package to your search list (e.g., via `library(MASS)`), you don't need to call `data`. Its datasets become available automatically when you attach it.

You can see a list of available datasets in `MASS`, or any other package, by using the `data` function with a `package` argument and no dataset name:

```
data(package = "pkgname")
```

See Also

See [Recipe 3.6](#) for more about the search path and [Recipe 3.8](#) for more about packages and the `library` function.

3.10 Installing Packages from CRAN

Problem

You found a package on CRAN, and now you want to install it on your computer.

Solution

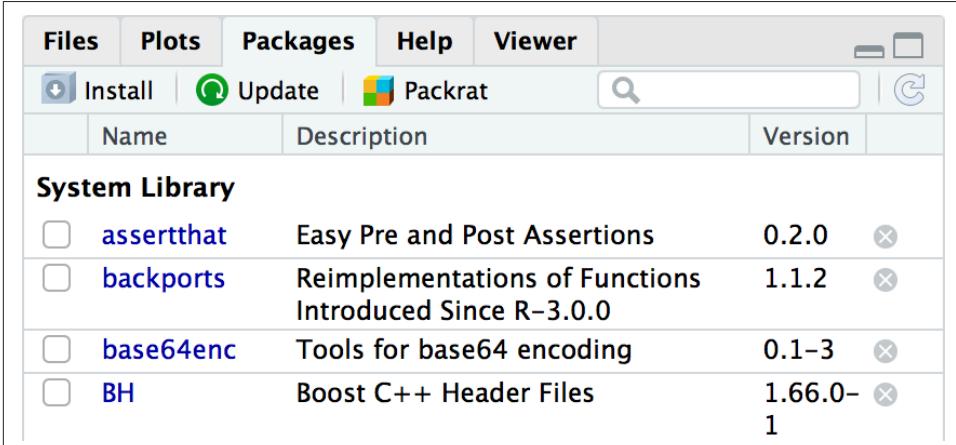
R code

Use the `install.packages` function, putting the name of the package in quotes:

```
install.packages("packagename")
```

RStudio

The Packages pane in RStudio helps make installing new R packages straightforward. All packages that are installed on your machine are listed in this pane, along with description and version information. To load a new package from CRAN, click on the Install button near the top of the Packages pane, shown in Figure 3-6.



The screenshot shows the RStudio interface with the 'Packages' tab selected in the top navigation bar. Below the navigation bar, there are three buttons: 'Install' (with a download icon), 'Update' (with a circular arrow icon), and 'Packrat' (with a cube icon). A search bar is also present. The main area is titled 'System Library' and lists four packages:

Name	Description	Version
assertthat	Easy Pre and Post Assertions	0.2.0
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.2
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.66.0-1

Figure 3-6. RStudio Packages pane

Discussion

Installing a package locally is the first step toward using it. If you are installing packages outside of RStudio, the installer may prompt you for a mirror site from which it can download the package files. It will then display a list of CRAN mirror sites. The top CRAN mirror is 0-Cloud. This is typically the best option, as it connects you to a globally mirrored *content delivery network* (CDN) sponsored by RStudio. If you want to select a different mirror, choose one geographically close to you.

The official CRAN server is a relatively modest machine generously hosted by the Department of Statistics and Mathematics at WU Wien, Vienna, Austria. If every R user downloaded from the official server, it would buckle under the load, so there are numerous mirror sites around the globe. In RStudio the default CRAN server is set to be the RStudio CRAN mirror. The RStudio CRAN mirror is accessible to all R users, not just those running the RStudio IDE.

If the new package depends upon other packages that are not already installed locally, then the R installer will automatically download and install those required packages. This is a huge benefit that frees you from the tedious task of identifying and resolving those dependencies.

There is a special consideration when you are installing on Linux or Unix. You can install the package either in the systemwide library or in your personal library. Packages in the systemwide library are available to everyone; packages in your personal library are (normally) used only by you. So, a popular, well-tested package would likely go in the systemwide library, whereas an obscure or untested package would go into your personal library.

By default, `install.packages` assumes you are performing a systemwide install. If you do not have sufficient user permissions to install in the systemwide library location, R will ask if you would like to install the package in a user library. The default that R suggests is typically a good choice. However, if you would like to control the path for your library location, you can use the `lib` argument of the `install.packages` function:

```
install.packages("packagename", lib = "~/lib/R")
```

Or you can change your default CRAN server as described in [Recipe 3.12](#).

See Also

See [Recipe 1.12](#) for ways to find relevant packages and [Recipe 3.8](#) for using a package after installing it.

See also [Recipe 3.12](#).

3.11 Installing a Package from GitHub

Problem

You've found an interesting package you'd like to try. However, the author has not yet published the package on CRAN, but has published it on GitHub. You'd like to install the package directly from GitHub.

Solution

Ensure you have the `devtools` package installed and loaded:

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

Then use `install_github` and the name of the GitHub repository to install directly from GitHub. For example, to install Thomas Lin Pederson's `tidygraph` package, you would execute the following:

```
install_github("thomasp85/tidygraph")
```

Discussion

The `devtools` package contains helper functions for installing R packages from remote repositories, like GitHub. If a package has been built as an R package and then hosted on GitHub, you can install the package using the `install_github` function by passing the GitHub username and repository name as a string parameter. You can determine the GitHub username and repo name from the GitHub URL, or from the top of the GitHub page, as in the example shown in [Figure 3-7](#).

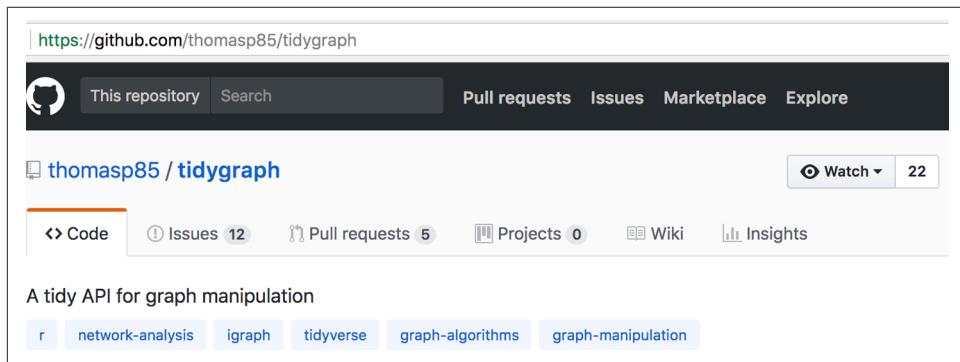


Figure 3-7. Example GitHub project page

3.12 Setting or Changing a Default CRAN Mirror

Problem

You are downloading packages. You want to set or change your default CRAN mirror.

Solution

In RStudio, you can change your default CRAN mirror from the RStudio Preferences menu shown in [Figure 3-8](#).

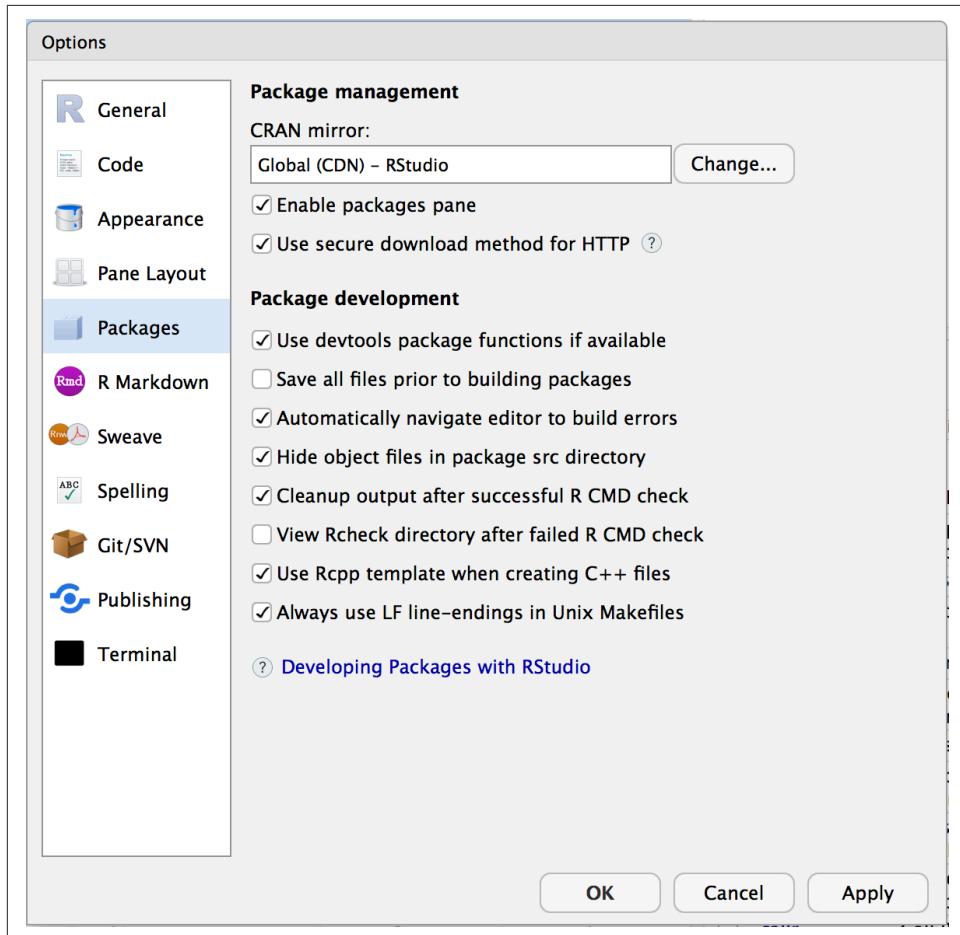


Figure 3-8. RStudio package preferences

If you are running R without RStudio, you can change your CRAN mirror using the following solution. This solution assumes you have an *.Rprofile*, as described in [Recipe 3.16](#):

1. Call the `chooseCRANmirror` function:

```
chooseCRANmirror()
```

R will present a list of CRAN mirrors.

2. Select a CRAN mirror from the list and press OK.

3. To get the URL of the mirror, look at the first element of the `repos` option:

```
options("repos")[[1]][1]
```

4. Add this line to your *.Rprofile* file. If you want the RStudio CRAN mirror, you would do the following:

```
options(repos = c(CRAN = "http://cran.rstudio.com"))
```

Or you could use the URL of another CRAN mirror.

Discussion

When you install packages, you probably use the same CRAN mirror each time (namely, the mirror closest to you or the RStudio mirror) because RStudio does not prompt you every time you load a package; it simply uses the setting from the Preferences menu. You may want to change that mirror to use a different mirror that's closer to you or controlled by your employer. Use this solution to change your repo so that every time you start R or RStudio, you will be using your desired repo.

The `repos` option is the name of your default mirror. The `chooseCRANmirror` function has the important side effect of setting the `repos` option according to your selection. The problem is that R forgets the setting when it exits, leaving no permanent default. By setting `repos` in your *.Rprofile*, you restore the setting every time R starts.

See Also

See [Recipe 3.16](#) for more about the *.Rprofile* file and the `options` function.

3.13 Running a Script

Problem

You captured a series of R commands in a text file. Now you want to execute them.

Solution

The `source` function instructs R to read the text file and execute its contents:

```
source("myScript.R")
```

Discussion

When you have a long or frequently used piece of R code, capture it inside a text file. That lets you easily rerun the code without having to retype it. Use the `source` function to read and execute the code, just as if you had typed it into the R console.

Suppose the file *hello.R* contains this one familiar greeting:

```
print("Hello, World!")
```

Then sourcing the file will execute the file's contents:

```
source("hello.R")
#> [1] "Hello, World!"
```

Setting `echo=TRUE` will echo the script's lines before they are executed, with the R prompt shown before each line:

```
source("hello.R", echo = TRUE)
#>
#> > print("Hello, World!")
#> [1] "Hello, World!"
```

See Also

See [Recipe 2.12](#) for running blocks of R code inside the GUI.

3.14 Running a Batch Script

Problem

You are writing a command script, such as a shell script in Unix or macOS or a BAT script in Windows. Inside your script, you want to execute an R script.

Solution

Run the R program with the `CMD BATCH` subcommand, giving the script name and the output filename:

```
R CMD BATCH scriptfile outputfile
```

If you want the output sent to `stdout` or if you need to pass command-line arguments to the script, consider the `Rscript` command instead:

```
Rscript scriptfile arg1 arg2 arg3
```

Discussion

R is normally an interactive program, one that prompts the user for input and then displays the results. Sometimes you want to run R in batch mode, reading commands from a script. This is especially useful inside shell scripts, such as scripts that include a statistical analysis.

The `CMD BATCH` subcommand puts R into batch mode, reading from `scriptfile` and writing to `outputfile`. It does not interact with a user.

You will likely use command-line options to adjust R's batch behavior to your circumstances. For example, using `--quiet` silences the startup messages that would otherwise clutter the output:

```
R CMD BATCH --quiet myScript.R results.out
```

Other useful options in batch mode include the following:

--slave

Like --quiet, but it makes R even more silent by inhibiting echo of the input.

--no-restore

At startup, do not restore the R workspace. This is important if your script expects R to begin with an empty workspace.

--no-save

At exit, do not save the R workspace. Otherwise, R will save its workspace and overwrite the *.RData* file in the working directory.

--no-init-file

Do not read either the *.Rprofile* or the *~/.Rprofile* file.

The **CMD BATCH** subcommand normally calls **proc.time** when your script completes, showing the execution time. If this annoys you, then end your script by calling the **q** function with **runLast=FALSE**, which will prevent the call to **proc.time**.

The **CMD BATCH** subcommand has two limitations: the output always goes to a file, and you cannot easily pass command-line arguments to your script. If either limitation is a problem, consider using the **Rscript** program that comes with R. The first command-line argument is the script name, and the remaining arguments are given to the script:

```
Rscript scriptfile.R arg1 arg2 arg3
```

Inside the script, you can access the command-line arguments by calling **commandArgs**, which returns the arguments as a vector of strings:

```
argv <- commandArgs(TRUE)
```

The **Rscript** program takes the same command-line options as **CMD BATCH**, which were just described.

Output is written to **stdout**, which R inherits from the calling shell script, of course. You can redirect the output to a file by using the normal redirection:

```
Rscript --slave scriptfile.R arg1 arg2 arg3 >results.out
```

Here is a small R script, *arith.R*, that takes two command-line arguments and performs four arithmetic operations on them:

```
argv <- commandArgs(TRUE)
x <- as.numeric(argv[1])
y <- as.numeric(argv[2])

cat("x =", x, "\n")
```

```
cat("y =", y, "\n")
cat("x + y = ", x + y, "\n")
cat("x - y = ", x - y, "\n")
cat("x * y = ", x * y, "\n")
cat("x / y = ", x / y, "\n")
```

The script is invoked like this:

```
Rscript arith.R 2 3.1415
```

which produces the following output:

```
x = 2
y = 3.1415
x + y = 5.1415
x - y = -1.1415
x * y = 6.283
x / y = 0.6366385
```

On Linux, Unix, or Mac, you can make the script fully self-contained by placing a `#!` line at the head with the path to the `Rscript` program. Suppose that `Rscript` is installed in `/usr/bin/Rscript` on your system. Adding this line to `arith.R` makes it a self-contained script:

```
#!/usr/bin/Rscript --slave

argv <- commandArgs(TRUE)
x <- as.numeric(argv[1])
.
. (etc.)
.
```

At the shell prompt, we mark the script as executable:

```
chmod +x arith.R
```

Now we can invoke the script directly without the `Rscript` prefix:

```
arith.R 2 3.1415
```

See Also

See [Recipe 3.13](#) for running a script from within R.

3.15 Locating the R Home Directory

Problem

You need to know the R home directory, which is where the configuration and installation files are kept.

Solution

R creates an environment variable called `R_HOME` that you can access by using the `Sys.getenv` function:

```
Sys.getenv("R_HOME")
#> [1] "/Library/Frameworks/R.framework/Resources"
```

Discussion

Most users will never need to know the R home directory. But system administrators or sophisticated users must know it in order to check or change the R installation files.

When R starts, it defines a system *environment* variable (not an R variable) called `R_HOME`, which is the path to the R home directory. The `Sys.getenv` function can retrieve the system environment variable value. Here are examples by platform. The exact value reported will almost certainly be different on your own computer:

- On Windows:

```
> Sys.getenv("R_HOME")
[1] "C:/PROGRA~1/R/R-34~1.4"
```

- On macOS:

```
> Sys.getenv("R_HOME")
[1] "/Library/Frameworks/R.framework/Resources"
```

- On Linux or Unix:

```
> Sys.getenv("R_HOME")
[1] "/usr/lib/R"
```

The Windows result looks funky because R reports the old, DOS-style compressed pathname. The full, user-friendly path would be `C:\Program Files\R\R-3.4.4` in this case.

On Unix and macOS, you can also run the R program from the shell and use the `RHOME` subcommand to display the home directory:

```
R RHOME
# /usr/lib/R
```

Note that the R home directory on Unix and macOS contains the installation files but not necessarily the R executable file. The executable could be in `/usr/bin` while the R home directory is, for example, `/usr/lib/R`.

3.16 Customizing R Startup

Problem

You want to customize your R sessions by, for instance, changing configuration options or preloading packages.

Solution

Create a script called *.Rprofile* that customizes your R session. R will execute the *.Rprofile* script when it starts. The placement of *.Rprofile* depends upon your platform:

macOS, Linux, or Unix

Save the file in your home directory (*~/.Rprofile*).

Windows

Save the file in your *Documents* directory.

Discussion

R executes profile scripts when it starts allowing you to tweak the R configuration options.

You can create a profile script called *.Rprofile* and place it in your home directory (macOS, Linux, Unix) or your *Documents* directory (Windows). The script can call functions to customize your sessions, such as this simple script that sets two environment variables and sets the console prompt to *R>*:

```
Sys.setenv(DB_USERID = "my_id")
Sys.setenv(DB_PASSWORD = "My_Password!")
options(prompt = "R> ")
```

The profile script executes in a bare-bones environment, so there are limits on what it can do. Trying to open a graphics window will fail, for example, because the *graphics* package is not yet loaded. Also, you should not attempt long-running computations.

You can customize a particular project by putting an *.Rprofile* file in the directory that contains the project files. When R starts in that directory, it reads the local *.Rprofile* file; this allows you to do project-specific customizations (e.g., setting your console prompt to a specific project name). However, if R finds a local profile, then it does *not* read the global profile. That can be annoying, but it's easily fixed: simply *source* the global profile from the local profile. On Unix, for instance, this local profile would execute the global profile first and then execute its local material:

```
source("~/Rprofile")
#
# ... remainder of local .Rprofile ...
#
```

Setting options

Some customizations are handled via calls to the `options` function, which sets the R configuration options. There are many such options, and the R help page for `options` lists them all:

```
help(options)
```

Here are some examples:

```
browser="path"
Path of default HTML browser
```

```
digits=n
Suggested number of digits to print when printing numeric values
```

```
editor="path"
Default text editor
```

```
prompt="string"
Input prompt
```

```
repos="url"
URL for default repository for packages
```

```
warn=n
Controls display of warning messages
```

Reproducibility

Many of us use certain packages over and over in our scripts (for example, the tidyverse packages). It is tempting to load these packages in your `.Rprofile` so that they are always available without you typing anything. As a matter of fact, this advice was given in the first edition of this book. However, the downside of loading packages in your `.Rprofile` is reproducibility. If someone else (or you, on another machine) tries to run your script, they may not realize that you had loaded packages in your `.Rprofile`. Your script might not work for them, depending on which packages *they* load. So while it might be convenient to load packages in `.Rprofile`, you will play better with collaborators (and your future self) if you explicitly call `library(packagename)` in your R scripts.

Another issue with reproducibility is when users change default behaviors of R inside their `.Rprofile`. An example of this would be setting `options(stringsAsFactors = FALSE)`. This is appealing, as many users would prefer this default. However, if some-

one runs the script without this option being set, they will get different results or not be able to run the script at all. This can lead to considerable frustration.

As a guideline, you should primarily put things in the `.Rprofile` that:

- Change the look and feel of R (e.g., `digits`).
- Are specific to your local environment (e.g., `browser`).
- Specifically need to be outside of your scripts (i.e., database passwords).
- Do not change the results of your analysis.

Startup sequence

Here is a simplified overview of what happens when R starts (type `help(Startup)` to see the full details):

1. **R executes the `Rprofile.site` script.** This is the site-level script that enables system administrators to override default options with localizations. The script's full path is `R_HOME/etc/Rprofile.site`. (`R_HOME` is the R home directory; see [Recipe 3.15](#).)
The R distribution does not include an `Rprofile.site` file. Rather, the system administrator creates one if it is needed.
2. **R executes the `.Rprofile` script in the working directory; or, if that file does not exist, executes the `.Rprofile` script in your home directory.** This is the user's opportunity to customize R for their own purposes. The `.Rprofile` script in the home directory is used for global customizations. The `.Rprofile` script in a lower-level directory can perform specific customizations when R is started there—for instance, customizing R when started in a project-specific directory.
3. **R loads the workspace saved in `.RData`, if that file exists in the working directory.** R saves your workspace in the file called `.RData` when it exits. It reloads your workspace from that file, restoring access to your local variables and functions. You can disable this behavior in RStudio through Tools → Global Options. We recommend you disable this option and always explicitly save and load your work.
4. **R executes the `.First` function, if you defined one.** The `.First` function is a useful place for users or projects to define startup initialization code. You can define it in your `.Rprofile` or in your workspace.
5. **R executes the `.First.sys` function.** This step loads the default packages. The function is internal to R and not normally changed by either users or administrators.

Note that R does not load the default packages until the final step, when it executes the `.First.sys` function. Before that, only the base package has been loaded. This is a key point, because it means the previous steps cannot assume that packages other than the base are available. It also explains why trying to open a graphics window in your `.Rprofile` script fails: the graphics packages aren't loaded yet.

See Also

See the R help page for `Startup` (`help(Startup)`) and the R help page for `options` (`help(options)`). See [Recipe 3.8](#) for more about loading packages.

3.17 Using R and RStudio in the Cloud

Problem

You want to run R and RStudio in a cloud environment.

Solution

The most straightforward way to use R in the cloud is to use the RStudio.cloud web service. To use the service, point your web browser to <http://rstudio.cloud> and set up an account, or log in with your Google or GitHub credentials.

Discussion

After you log in, click New Project to begin a new RStudio session in a new workspace. You'll be greeted by the familiar RStudio interface shown in [Figure 3-9](#).

Keep in mind that as of this writing the RStudio.cloud service is in alpha testing and may not be 100% stable. Your work will persist after you log off. However, as with any system, it is a good idea to ensure you have backups of all the work you do. A common work pattern is to connect your project in RStudio.cloud to a GitHub repository and push your changes frequently from RStudio.cloud to GitHub. This workflow has been used significantly in the writing of this book.

Use of Git and GitHub is beyond the scope of this book, but if you are interested in learning more, we highly recommend Jenny Bryan's web book [*Happy Git and GitHub for the useR*](#).

In its current alpha state, RStudio.cloud limits each session to 1 GB of RAM and 3 GB of drive space—so it's a great platform for learning and teaching but might not (yet) be the platform on which you want to build a commercial data science laboratory. RStudio has expressed its intent to offer greater processing power and storage as part of a paid tier of service as the platform matures.

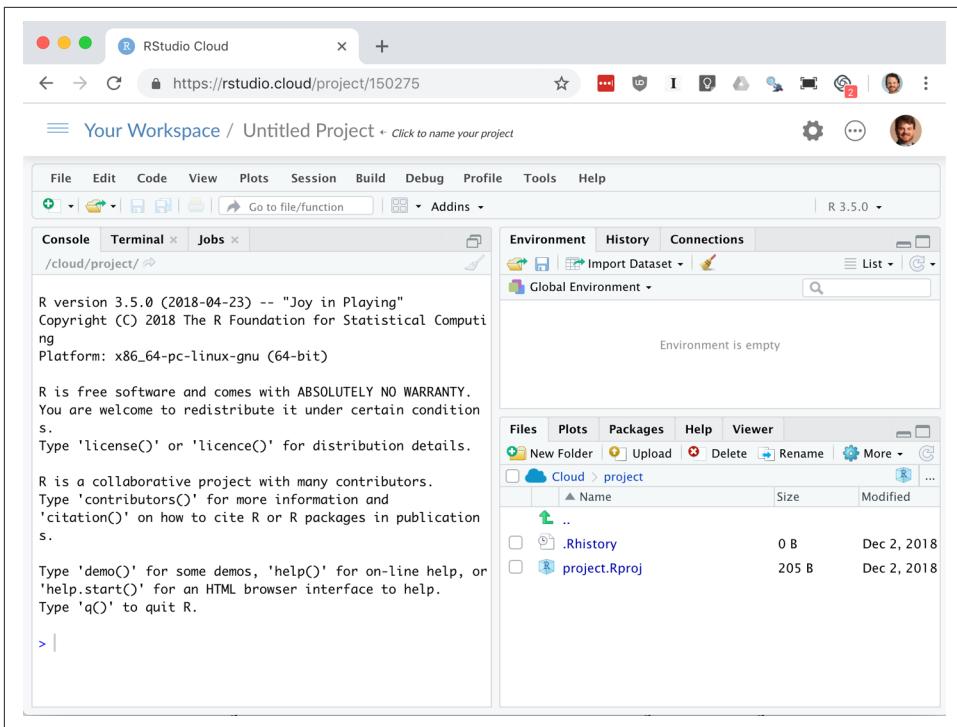


Figure 3-9. RStudio.cloud

If you need more computing power than offered by RStudio.cloud and you are willing to pay for the services, both [Amazon Web Services \(AWS\)](#) and [Google Cloud Platform](#) offer cloud-based RStudio offerings. Other cloud platforms that support Docker, such as [Digital Ocean](#), are also reasonable options for cloud-hosted RStudio.

CHAPTER 4

Input and Output

All statistical work begins with data, and most data is stuck inside files and databases. Dealing with input is probably the first step of implementing any significant statistical project.

All statistical work ends with reporting numbers back to a client, even if you are the client. Formatting and producing output is probably the climax of your project.

Casual R users can solve their input problems by using basic `readr` package functions such as `read_csv` to read CSV files and `read_delim` to read more complicated, tabular data. They can use `print`, `cat`, and `format` to produce simple reports.

Users with heavy-duty input/output (I/O) needs are strongly encouraged to read the R Data Import/Export guide, [available on CRAN](#). This manual includes important information on reading data from sources such as spreadsheets, binary files, other statistical systems, and relational databases.

4.1 Entering Data from the Keyboard

Problem

You have a small amount of data—too small to justify the overhead of creating an input file. You just want to enter the data directly into your workspace.

Solution

For very small datasets, enter the data as literals using the `c` constructor for vectors:

```
scores <- c(61, 66, 90, 88, 100)
```

Discussion

When working on a simple problem, you may not want the hassle of creating and then reading a data file outside of R. You may just want to enter the data into R. The easiest way to do so is by using the `c` constructor for vectors, as shown in the Solution.

You can use this approach for data frames, too, by entering each variable (column) as a vector:

```
points <- data.frame(  
  label = c("Low", "Mid", "High"),  
  lbound = c(0, 0.67, 1.64),  
  ubound = c(0.67, 1.64, 2.33)  
)
```

See Also

For cutting and pasting data from another application into R, be sure to look at [data pasta](#), a package that provides RStudio add-ins that make pasting data into your scripts easier.

4.2 Printing Fewer Digits (or More Digits)

Problem

Your output contains too many digits, or too few digits. You want to print fewer, or more.

Solution

For `print`, the `digits` parameter can control the number of printed digits.

For `cat`, use the `format` function (which also has a `digits` parameter) to alter the formatting of numbers.

Discussion

R normally formats floating-point output to have seven digits. This works well most of the time but can become annoying when you have lots of numbers to print in a small space. It gets downright misleading when there are only a few significant digits in your numbers and R still prints seven.

The `print` function lets you vary the number of printed digits using the `digits` parameter:

```

print(pi, digits = 4)
#> [1] 3.142
print(100 * pi, digits = 4)
#> [1] 314.2

```

The `cat` function does not give you direct control over formatting. Instead, use the `format` function to format your numbers before calling `cat`:

```

cat(pi, "\n")
#> 3.14
cat(format(pi, digits = 4), "\n")
#> 3.142

```

This is R, so both `print` and `format` will format entire vectors at once:

```

print(pnorm(-3:3), digits = 2)
#> [1] 0.0013 0.0228 0.1587 0.5000 0.8413 0.9772 0.9987
format(pnorm(-3:3), digits = 2)
#> [1] "0.0013" "0.0228" "0.1587" "0.5000" "0.8413" "0.9772" "0.9987"

```

Notice that both `print` and `format` format the vector elements consistently, finding the number of significant digits necessary to format the smallest number and then formatting all numbers to have the same width (though not necessarily the same number of digits). This is extremely useful for formatting an entire table:

```

q <- seq(from = 0, to = 3, by = 0.5)
tbl <- data.frame(Quant = q,
                  Lower = pnorm(-q),
                  Upper = pnorm(q))
tbl                                         # Unformatted print
#>   Quant    Lower   Upper
#> 1   0.0  0.50000  0.500
#> 2   0.5  0.30854  0.691
#> 3   1.0  0.15866  0.841
#> 4   1.5  0.06681  0.933
#> 5   2.0  0.02275  0.977
#> 6   2.5  0.00621  0.994
#> 7   3.0  0.00135  0.999
print(tbl, digits = 2)                      # Formatted print: fewer digits
#>   Quant    Lower   Upper
#> 1   0.0  0.5000  0.50
#> 2   0.5  0.3085  0.69
#> 3   1.0  0.1587  0.84
#> 4   1.5  0.0668  0.93
#> 5   2.0  0.0228  0.98
#> 6   2.5  0.0062  0.99
#> 7   3.0  0.0013  1.00

```

As you can see, when an entire vector or column is formatted, each element in the vector or column is formatted the same way.

You can also alter the format of *all* output by using the `options` function to change the default for `digits`:

```
pi
#> [1] 3.14
options(digits = 15)
pi
#> [1] 3.14159265358979
```

But this is a poor choice in our experience, since it also alters the output from R's built-in functions, and that alteration will likely be unpleasant.

See Also

Other functions for formatting numbers include `sprintf` and `formatC`; see their help pages for details.

4.3 Redirecting Output to a File

Problem

You want to redirect the output from R to a file instead of your console.

Solution

You can redirect the output of the `cat` function by using its `file` argument:

```
cat("The answer is", answer, "\n", file = "filename.txt")
```

Use the `sink` function to redirect *all* output from both `print` and `cat`. Call `sink` with a `filename` argument to begin redirecting console output to that file. When you are done, use `sink` with no argument to close the file and resume output to the console:

```
sink("filename")           # Begin writing output to file
# ... other session work ...
sink()                     # Resume writing output to console
```

Discussion

The `print` and `cat` functions normally write their output to your console. The `cat` function writes to a file if you supply a `file` argument, which can be either a filename or a connection. The `print` function cannot redirect its output, but the `sink` function can force all output to a file. A common use for `sink` is to capture the output of an R script:

```
sink("script_output.txt")    # Redirect output to file
source("script.R")          # Run the script, capturing its output
sink()                      # Resume writing output to console
```

If you are repeatedly `cat`ing items to one file, be sure to set `append=TRUE`. Otherwise, each call to `cat` will simply overwrite the file's contents:

```
cat(data, file = "analysisReport.out")
cat(results, file = "analysisReport.out", append = TRUE)
cat(conclusion, file = "analysisReport.out", append = TRUE)
```

Hardcoding filenames like this is a tedious and error-prone process. Did you notice that the filename is misspelled in the second line? Instead of hardcoding the filename repeatedly, we suggest opening a connection to the file and writing your output to the connection:

```
con <- file("analysisReport.out", "w")
cat(data, file = con)
cat(results, file = con)
cat(conclusion, file = con)
close(con)
```

(You don't need `append=TRUE` when writing to a connection because `append` is the default with connections.) This technique is especially valuable inside R scripts because it makes your code more reliable and more maintainable.

4.4 Listing Files

Problem

You want an R vector that is a listing of the files in your working directory.

Solution

The `list.files` function shows the contents of your working directory:

```
list.files()
#> [1] "_book"                      "_bookdown_files"
#> [3] "_bookdown.yml"                "_common.R"
#> [5] "_main.log"                   "_main.rds"
#> [7] ".output.yml"                 "01_GettingStarted_cache"
#> [9] "01_GettingStarted.md"        "01_GettingStarted.Rmd"
#> # etc.
```

Discussion

This function is terribly handy to grab the names of all files in a subdirectory. You can use it to refresh your memory of your filenames or, more likely, as input into another process, like importing data files.

You can pass `list.files` a path and a pattern to show files in a specific path and matching a specific regular expression pattern:

```

list.files(path = 'data/') # show files in a directory
#> [1] "ac.rdata"           "adf.rdata"
#> [3] "anova.rdata"       "anova2.rdata"
#> [5] "bad.rdata"          "batches.rdata"
#> [7] "bnd_cmtv.Rdata"    "compositePerf-2010.csv"
#> [9] "conf.rdata"         "daily.prod.rdata"
#> [11] "data1.csv"         "data2.csv"
#> [13] "datafile_missing.tsv" "datafile.csv"
#> [15] "datafile.fwf"      "datafile.qsv"
#> [17] "datafile.ssv"      "datafile.tsv"
#> [19] "datafile1.ssv"     "df_decay.rdata"
#> [21] "df_squared.rdata"   "diffs.rdata"
#> [23] "example1_headless.csv" "example1.csv"
#> [25] "excel_table_data.xlsx" "get_USDA_NASS_data.R"
#> [27] "ibm.rdata"          "iris_excel.xlsx"
#> [29] "lab_df.rdata"       "movies.sas7bdat"
#> [31] "nacho_data.csv"     "NearestPoint.R"
#> [33] "not_a_csv.txt"      "opt.rdata"
#> [35] "outcome.rdata"      "pca.rdata"
#> [37] "pred.rdata"         "pred2.rdata"
#> [39] "sat.rdata"          "singles.txt"
#> [41] "state_corn_yield.rds" "student_data.rdata"
#> [43] "suburbs.txt"        "tab1.csv"
#> [45] "tls.rdata"          "triples.txt"
#> [47] "ts_acf.rdata"       "workers.rdata"
#> [49] "world_series.csv"    "xy.rdata"
#> [51] "yield.Rdata"         "z.RData"
list.files(path = 'data/', pattern = '\\.csv')
#> [1] "compositePerf-2010.csv" "data1.csv"
#> [3] "data2.csv"              "datafile.csv"
#> [5] "example1_headless.csv" "example1.csv"
#> [7] "nacho_data.csv"        "tab1.csv"
#> [9] "world_series.csv"

```

To see all the files in your subdirectories, too, use:

```
list.files(recursive = T)
```

A possible “gotcha” of `list.files` is that it ignores hidden files—typically, any file whose name begins with a dot. If you don’t see the file you expected to see, try setting `all.files=TRUE`:

```

list.files(path = 'data/', all.files = TRUE)
#> [1] "."                  ".."
#> [3] ".DS_Store"          ".hidden_file.txt"
#> [5] "ac.rdata"            "adf.rdata"
#> [7] "anova.rdata"         "anova2.rdata"
#> [9] "bad.rdata"           "batches.rdata"
#> [11] "bnd_cmtv.Rdata"    "compositePerf-2010.csv"
#> [13] "conf.rdata"         "daily.prod.rdata"
#> [15] "data1.csv"          "data2.csv"
#> [17] "datafile_missing.tsv" "datafile.csv"
#> [19] "datafile.fwf"       "datafile.qsv"

```

```

#> [21] "datafile.csv"           "datafile.tsv"
#> [23] "datafile1.csv"          "df_decay.rdata"
#> [25] "df_squared.rdata"       "diffs.rdata"
#> [27] "example1_headless.csv"   "example1.csv"
#> [29] "excel_table_data.xlsx"   "get_USDA_NASS_data.R"
#> [31] "ibm.rdata"              "iris_excel.xlsx"
#> [33] "lab_df.rdata"           "movies.sas7bdat"
#> [35] "nacho_data.csv"         "NearestPoint.R"
#> [37] "not_a_csv.txt"          "opt.rdata"
#> [39] "outcome.rdata"          "pca.rdata"
#> [41] "pred.rdata"             "pred2.rdata"
#> [43] "sat.rdata"              "singles.txt"
#> [45] "state_corn_yield.rds"   "student_data.rdata"
#> [47] "suburbs.txt"            "tab1.csv"
#> [49] "tls.rdata"              "triples.txt"
#> [51] "ts_acf.rdata"           "workers.rdata"
#> [53] "world_series.csv"        "xy.rdata"
#> [55] "yield.Rdata"             "z.RData"

```

If you just want to see which files are in a directory and not use the filenames in a procedure, the easiest way is to open the Files pane in the lower-right corner of RStudio. But keep in mind that the RStudio Files pane hides files that start with a dot, as you can see in [Figure 4-1](#).

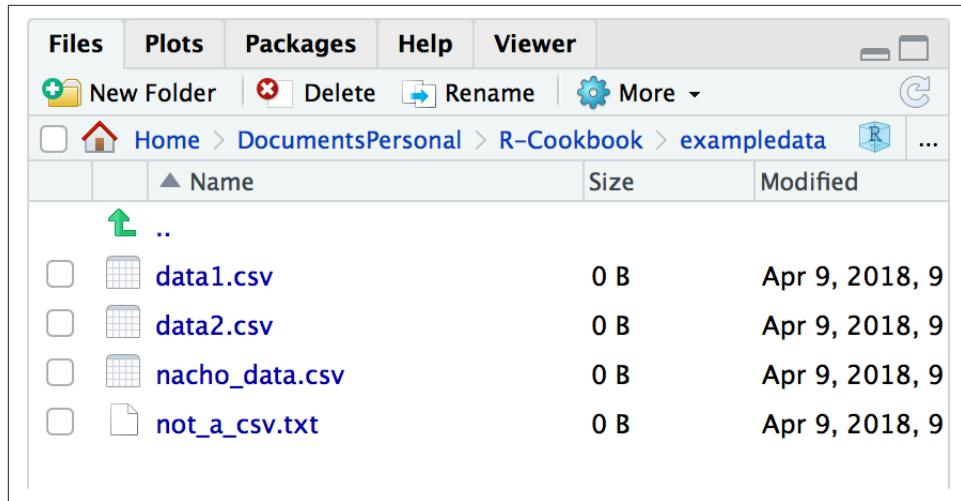


Figure 4-1. RStudio Files pane

See Also

R has other handy functions for working with files; see `help(files)`.

4.5 Dealing with “Cannot Open File” in Windows

Problem

You are running R on Windows, and you are using filenames such as `C:\data\sample.txt`. R says it cannot open a file, but you know the file does exist.

Solution

The backslashes in the filepath are causing trouble. You can solve this problem in one of two ways:

- Change the backslashes to forward slashes: `"C:/data/sample.txt"`.
- Double the backslashes: `"C:\\data\\\\sample.txt"`.

Discussion

When you open a file in R, you give the filename as a character string. Problems arise when the name contains backslashes (`\`) because backslashes have a special meaning inside strings. You’ll probably get something like this:

```
samp <- read_csv("C:\\Data\\sample-data.csv")
#> Error: '|D' is an unrecognized escape in character string starting "'C:\\D"
```

R escapes every character that follows a backslash and then removes the backslashes. That leaves a meaningless filepath, such as `C:Datasample-data.csv` in this example.

The simple solution is to use forward slashes instead of backslashes. R leaves the forward slashes alone, and Windows treats them just like backslashes. Problem solved:

```
samp <- read_csv("C:/Data/sample-data.csv")
```

An alternative solution is to double the backslashes, since R replaces two consecutive backslashes with a single backslash:

```
samp <- read_csv("C:\\\\Data\\\\sample-data.csv")
```

4.6 Reading Fixed-Width Records

Problem

You are reading data from a file of fixed-width records: records whose data items occur at fixed boundaries.

Solution

Use the `read_fwf` function from the `readr` package (which is part of the tidyverse). The main arguments are the filename and the description of the fields:

```
library(tidyverse)
records <- read_fwf("myfile.txt",
                     fwf_cols(col1 = 10,
                               col2 = 7))
records
```

This form uses the `fwf_cols` parameter to pass column names and widths to the function. You can also pass column parameters in other ways, as discussed next.

Discussion

For reading data into R, we highly recommend the `readr` package. There are Base R functions for reading in text files, but `readr` improves on these base functions with faster performance, better defaults, and more flexibility.

Suppose we want to read an entire file of fixed-width records, such as *fixed-width.txt*, shown here:

Fisher	R.A.	1890	1962
Pearson	Karl	1857	1936
Cox	Gertrude	1900	1978
Yates	Frank	1902	1994
Smith	Kirstine	1878	1939

We need to know the column widths. In this case the columns are:

- Last name, 10 characters
- First name, 10 characters
- Year of birth, 5 characters
- Year of death, 5 characters

There are five different ways to define the columns using `read_fwf`. Pick the one that's easiest to use (or remember) in your situation:

- `read_fwf` can try to guess your column widths if there is empty space between the columns, with the `fwf_empty` option:

```
file <- "./data/datafile.fwf"
t1 <- read_fwf(file,
               fwf_empty(file,
                         col_names = c("last", "first", "birth", "death")))
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
```

```

#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )

```

- You can define each column by a vector of widths followed by a vector of names with `fwf_widths`:

```

t2 <- read_fwf(file, fwf_widths(c(10, 10, 5, 4),
                                c("last", "first", "birth", "death")))
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )

```

- The columns can be defined with `fwf_cols`, which takes a series of column names followed by the column widths:

```

t3 <-
read_fwf("./data/datafile.fwf",
      fwf_cols(
        last = 10,
        first = 10,
        birth = 5,
        death = 5
      ))
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )

```

- Each column can be defined by a beginning position and ending position with `fwf_cols`:

```

t4 <- read_fwf(file, fwf_cols(
  last = c(1, 10),
  first = c(11, 20),
  birth = c(21, 25),
  death = c(26, 30)
))
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),

```

```

#>   death = col_double()
#> )

• You can also define the columns with a vector of starting positions, a vector of
ending positions, and a vector of column names, with fwf_positions:

t5 <- read_fwf(file, fwf_positions(
  c(1, 11, 21, 26),
  c(10, 20, 25, 30),
  c("first", "last", "birth", "death")
))
#> Parsed with column specification:
#> cols(
#>   first = col_character(),
#>   last = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )

```

The `read_fwf` function returns a *tibble*, which is a tidyverse flavor of data frame. As is common with tidyverse packages, `read_fwf` has a good selection of default assumptions that make it less tricky to use than some Base R functions for importing data. For example, `read_fwf` will, by default, import character fields as characters, not factors, which prevents much pain and consternation for users.

See Also

See [Recipe 4.7](#) for more discussion of reading text files.

4.7 Reading Tabular Data Files

Problem

You want to read a text file that contains a table of whitespace-delimited data.

Solution

Use the `read_table2` function from the `readr` package, which returns a tibble:

```

library(tidyverse)

tab1 <- read_table2("./data/datafile.tsv")
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )

```

```
tab1
#> # A tibble: 5 x 4
#>   last     first    birth   death
#>   <chr>   <chr>    <dbl>   <dbl>
#> 1 Fisher   R.A.     1890    1962
#> 2 Pearson  Karl     1857    1936
#> 3 Cox      Gertrude  1900    1978
#> 4 Yates   Frank     1902    1994
#> 5 Smith   Kirstine  1878    1939
```

Discussion

Tabular data files are quite common. They are text files with a simple format:

- Each line contains one record.
- Within each record, fields (items) are separated by a whitespace delimiter, such as a space or tab.
- Each record contains the same number of fields.

This format is more free-form than the fixed-width format because fields needn't be aligned by position. Here is the data file from [Recipe 4.6](#) in tabular format, using a tab character between fields:

```
last     first    birth   death
Fisher   R.A.     1890    1962
Pearson  Karl     1857    1936
Cox      Gertrude  1900    1978
Yates   Frank     1902    1994
Smith   Kirstine  1878    1939
```

The `read_table2` function is designed to make some good guesses about your data. It assumes your data has column names in the first row, it guesses your delimiter, and it imputes your column types based on the first 1,000 records in your dataset. Next is an example with space-delimited data.

The source file looks like this:

```
last first birth death
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

And `read_table2` makes some rational guesses:

```
t <- read_table2("./data/datafile1.csv")
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
```

```

#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )
print(t)
#> # A tibble: 5 x 4
#>   last    first    birth  death
#>   <chr>   <chr>   <dbl> <dbl>
#> 1 Fisher  R.A.     1890  1962
#> 2 Pearson Karl     1857  1936
#> 3 Cox      Gertrude 1900  1978
#> 4 Yates   Frank    1902  1994
#> 5 Smith   Kirstine 1878  1939

```

`read_table2` often guesses correctly. But as with other `readr` import functions, you can overwrite the defaults with explicit parameters:

```

t <-
  read_table2(
    "./data/datafile1.csv",
    col_types = c(
      col_character(),
      col_character(),
      col_integer(),
      col_integer()
    )
  )

```

If any field contains the string "NA", then `read_table2` assumes that the value is missing and converts it to NA. Your data file might employ a different string to signal missing values, in which case use the `na` parameter. The SAS convention, for example, is that missing values are signaled by a single period (.). We can read such text files using the `na=".."` option. If we have a file named *datafile_missing.tsv* that has a missing value indicated with a . in the last row:

last	first	birth	death
Fisher	R.A.	1890	1962
Pearson	Karl	1857	1936
Cox	Gertrude	1900	1978
Yates	Frank	1902	1994
Smith	Kirstine	1878	1939
Cox	David	1924	.

we can import it like so:

```

t <- read_table2("./data/datafile_missing.tsv", na = ".")
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()

```

```

#> )
t
#> # A tibble: 6 x 4
#>   last    first    birth  death
#>   <chr>   <chr>   <dbl> <dbl>
#> 1 Fisher  R.A.     1890  1962
#> 2 Pearson Karl     1857  1936
#> 3 Cox     Gertrude  1900  1978
#> 4 Yates   Frank    1902  1994
#> 5 Smith   Kirstine 1878  1939
#> 6 Cox     David    1924    NA

```

We're huge fans of *self-describing* data: data files that describe their own contents. (A computer scientist would say the file contains its own metadata.) The `read_table2` function makes the default assumption that the first line of your file contains a header line with column names. If your file does not have column names, you can turn this off with the parameter `col_names = FALSE`.

An additional type of metadata supported by `read_table2` is comment lines. Using the `comment` parameter you can tell `read_table2` which character distinguishes comment lines. The following file has a comment line at the top that starts with #:

```

# The following is a list of statisticians
last first birth death
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939

```

so we can import this file as follows:

```

t <- read_table2("./data/datafile.csv", comment = '#')
#> Parsed with column specification:
#> cols(
#>   last = col_character(),
#>   first = col_character(),
#>   birth = col_double(),
#>   death = col_double()
#> )
t
#> # A tibble: 5 x 4
#>   last    first    birth  death
#>   <chr>   <chr>   <dbl> <dbl>
#> 1 Fisher  R.A.     1890  1962
#> 2 Pearson Karl     1857  1936
#> 3 Cox     Gertrude  1900  1978
#> 4 Yates   Frank    1902  1994
#> 5 Smith   Kirstine 1878  1939

```

`read_table2` has many parameters for controlling how it reads and interprets the input file. See the help page (`?read_table2`) or the `readr` vignette

(vignette("readr")) for more details. If you're curious about the difference between `read_table` and `read_table2`, it's in the help file... but the short answer is that `read_table` is slightly less forgiving in file structure and line length.

See Also

If your data items are separated by commas, see [Recipe 4.8](#) for reading a CSV file.

4.8 Reading from CSV Files

Problem

You want to read data from a comma-separated values (CSV) file.

Solution

The `read_csv` function from the `readr` package is a fast (and, according to the documentation, fun) way to read CSV files. If your CSV file has a header line, use this:

```
library(tidyverse)  
  
tbl <- read_csv("datafile.csv")
```

If your CSV file does not contain a header line, set the `col_names` option to `FALSE`:

```
tbl <- read_csv("datafile.csv", col_names = FALSE)
```

Discussion

The CSV file format is popular because many programs can import and export data in that format. This includes R, Excel, other spreadsheet programs, many database managers, and most statistical packages. A CSV file is a flat file of tabular data, where each line in the file is a row of data, and each row contains data items separated by commas. Here is a very simple CSV file with three rows and three columns. The first line is a header line that contains the column names, also separated by commas:

```
label,lbound,ubound  
low,0,0.674  
mid,0.674,1.64  
high,1.64,2.33
```

The `read_csv` function reads the data and creates a tibble. The function assumes that your file has a header line unless told otherwise:

```
tbl <- read_csv("./data/example1.csv")  
#> Parsed with column specification:  
#> cols(  
#>   label = col_character(),  
#>   lbound = col_double(),
```

```

#>   ubound = col_double()
#> )
tbl
#> # A tibble: 3 x 3
#>   label lbound ubound
#>   <chr>  <dbl>  <dbl>
#> 1 low     0      0.674
#> 2 mid     0.674  1.64
#> 3 high    1.64   2.33

```

Observe that `read_csv` took the column names from the header line for the tibble. If the file did not contain a header, then we would specify `col_names=FALSE` and R would synthesize column names for us (`X1`, `X2`, and `X3` in this case):

```

tbl <- read_csv("./data/example1.csv", col_names = FALSE)
#> Parsed with column specification:
#> cols(
#>   X1 = col_character(),
#>   X2 = col_character(),
#>   X3 = col_character()
#> )
tbl
#> # A tibble: 4 x 3
#>   X1     X2     X3
#>   <chr> <chr> <chr>
#> 1 label lbound ubound
#> 2 low     0      0.674
#> 3 mid     0.674  1.64
#> 4 high    1.64   2.33

```

Sometimes it's convenient to put metadata in files. If this metadata starts with a common character, such as a pound sign (#), we can use the `comment=FALSE` parameter to ignore metadata lines.

The `read_csv` function has many useful bells and whistles. A few of these options and their default values include:

`na = c("", "NA")`

Indicates what values represent missing or NA values

`comment = ""`

Indicates which lines to ignore as comments or metadata

`trim_ws = TRUE`

Indicates whether to drop whitespace at the beginning and/or end of fields

`skip = 0`

Indicates the number of rows to skip at the beginning of the file

`guess_max = min(1000, n_max)`

Indicates the number of rows to consider when imputing column types

See the R help page, `help(read_csv)`, for more details on all the available options.

If you have a data file that uses semicolons (;) for separators and commas for the decimal mark, as is common outside of North America, you should use the function `read_csv2`, which is built for that very situation.

See Also

See [Recipe 4.9](#). See also the vignette for `readr: vignette(readr)`.

4.9 Writing to CSV Files

Problem

You want to save a matrix or data frame in a file using the comma-separated values format.

Solution

The `write_csv` function from the tidyverse `readr` package can write a CSV file:

```
library(tidyverse)  
  
write_csv(df, path = "outfile.csv")
```

Discussion

The `write_csv` function writes tabular data to an ASCII file in CSV format. Each row of data creates one line in the file, with data items separated by commas (,). We can start with the data frame `tab1` we created previously in [Recipe 4.7](#):

```
library(tidyverse)  
  
write_csv(tab1, "./data/tab1.csv")
```

This example creates a file called `tab1.csv` in the `data` directory, which is a subdirectory of the current working directory. The file looks like this:

```
last,first,birth,death  
Fisher,R.A.,1890,1962  
Pearson,Karl,1857,1936  
Cox,Gertrude,1900,1978  
Yates,Frank,1902,1994  
Smith,Kirstine,1878,1939
```

`write_csv` has a number of parameters with typically very good defaults. Should you want to adjust the output, here are a few parameters you can change, along with their defaults:

```
col_names = TRUE
```

Indicates whether or not the first row contains column names.

```
col_types = NULL
```

`write_csv` will look at the first 1,000 rows (changeable with `guess_max`) and make an informed guess as to what data types to use for the columns. If you'd rather explicitly state the column types, you can do so by passing a vector of column types to the parameter `col_types`.

```
na = c("", "NA")
```

Indicates what values represent missing or NA values.

```
comment = ""
```

Indicates which lines to ignore as comments or metadata.

```
trim_ws = TRUE
```

Indicates whether to drop whitespace at the beginning and/or end of fields.

```
skip = 0
```

Indicates the number of rows to skip at the beginning of the file.

```
guess_max = min(1000, n_max)
```

Indicates the number of rows to consider when guessing column types.

See Also

See [Recipe 3.1](#) for more about the current working directory and [Recipe 4.18](#) for other ways to save data to files. For more info on reading and writing text files, see the `readr` vignette: `vignette(readr)`.

4.10 Reading Tabular or CSV Data from the Web

Problem

You want to read data directly from the web into your R workspace.

Solution

Use the `read_csv` or `read_table2` functions from the `readr` package, using a URL instead of a filename. The functions will read directly from the remote server:

```
library(tidyverse)

berkley <- read_csv('http://bit.ly/barkley18', comment = '#')
#> Parsed with column specification:
#> cols(
#>   Name = col_character(),
```

```
#>   Location = col_character(),
#>   Time = col_time(format = "")
#> )
```

You can also open a connection using the URL and then read from the connection, which may be preferable for complicated files.

Discussion

The web is a gold mine of data. You could download the data into a file and then read the file into R, but it's more convenient to read directly from the web. Give the URL to `read_csv`, `read_table2`, or another `read` function in `readr` (depending upon the format of the data), and the data will be downloaded and parsed for you. No fuss, no muss.

Aside from using a URL, this recipe is just like reading from a CSV file (see [Recipe 4.8](#)) or a complex file ([Recipe 4.15](#)), so all the comments in those recipes apply here, too.

Remember that URLs work for FTP servers, not just HTTP servers. This means that R can also read data from FTP sites using URLs:

```
tbl <- read_table2("ftp://ftp.example.com/download/data.txt")
```

See Also

See [Recipe 4.8](#) and [Recipe 4.15](#).

4.11 Reading Data from Excel

Problem

You want to read data in from an Excel file.

Solution

The `openxlsx` package makes reading Excel files easy:

```
library(openxlsx)
df1 <- read.xlsx(xlsxFile = "file.xlsx",
                  sheet = 'sheet_name')
```

Discussion

The package `openxlsx` is a good choice for both reading and writing Excel files with R. If we're reading in an entire sheet, using the `read.xlsx` function is a simple option. We need only pass in a filename and, if desired, the name of the sheet we want imported:

```

library(openxlsx)

df1 <- read.xlsx(xlsxFile = "data/iris_excel.xlsx",
                  sheet = 'iris_data')
head(df1, 3)
#>   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

```

But `openxlsx` supports more complex workflows.

A common pattern is to read a named table out of an Excel file and into an R data frame. This is trickier because the sheet we're reading from may have values outside of the named table, but we want to only read in the named table range. We can use the functions in `openxlsx` to get the location of a table, then read that range of cells into a data frame.

First we load the entire workbook into R:

```

library(openxlsx)
wb <- loadWorkbook("data/excel_table_data.xlsx")

```

Then we can use the `getTables` function to get the names and ranges of all the Excel tables in the `input_data` sheet and select the one table we want. In this example the Excel table we are after is named `example_table`:

```

tables <- getTables(wb, 'input_data')
table_range_str <- names(tables[tables == 'example_table'])
table_range_refs <- strsplit(table_range_str, ':')[[1]]

# use a regex to extract out the row numbers
table_range_row_num <- gsub("[^0-9.]", "", table_range_refs)

# extract out the column numbers
table_range_col_num <- convertFromExcelRef(table_range_refs)

```

Now the vector `table_range_col_num` contains the column numbers of our named table, while `table_range_row_num` contains the row numbers of our named table. We can then use the `read.xlsx` function to pull in only the rows and columns we are after:

```

df <- read.xlsx(
  xlsxFile = "data/excel_table_data.xlsx",
  sheet = 'input_data',
  cols = table_range_col_num[1]:table_range_col_num[2],
  rows = table_range_row_num[1]:table_range_row_num[2]
)

```

While this may seem complicated, this design pattern can save a lot of hassle when sharing data with analysts who are using highly structured Excel files that include named tables.

See Also

You can see the vignette for `openxlsx` by installing `openxlsx` and running `vignette('Introduction', package='openxlsx')`.

The `readxl` package is part of the tidyverse and provides fast, simple reading of Excel files. However, `readxl` does not currently support named Excel tables.

The `writexl` package is a fast and lightweight (no dependencies) package for writing Excel files (discussed in [Recipe 4.12](#)).

4.12 Writing a Data Frame to Excel

Problem

You want to write an R data frame to an Excel file.

Solution

The `openxlsx` package makes writing to Excel files relatively easy. While there are lots of options in `openxlsx`, a typical pattern is to specify an Excel filename and a sheet name:

```
library(openxlsx)
write.xlsx(df,
           sheetName = "some_sheet",
           file = "out_file.xlsx")
```

Discussion

The `openxlsx` package has a huge number of options for controlling many aspects of the Excel object model. We can use it to set cell colors, define named ranges, and set cell outlines, for example. It also has a few helper functions like `write.xlsx` that make simple tasks super easy.

When businesses work with Excel, it's a good practice to keep all input data in an Excel file in a named Excel table, which makes accessing the data easier and less error prone. However, if you use `openxlsx` to overwrite an Excel table in one of the sheets, you run the risk that the new data may contain fewer rows than the Excel table it replaces. That could cause errors, as you would end up with old data and new data in contiguous rows. The solution is to first delete the existing Excel table, then add the new data back into the same location and assign the new data to a named Excel table. To do this we need to use the more advanced Excel manipulation features of `openxlsx`.

First we use `loadWorkbook` to read the Excel workbook into R in its entirety:

```
library(openxlsx)

wb <- loadWorkbook("data/excel_table_data.xlsx")
```

Before we delete the table, we want to extract the table's starting row and column:

```
tables <- getTables(wb, 'input_data')
table_range_str <- names(tables[tables == 'example_table'])
table_range_refs <- strsplit(table_range_str, ':')[[1]]

# use a regex to extract out the starting row number
table_row_num <- gsub("[^0-9]", "", table_range_refs)[[1]]

# extract out the starting column number
table_col_num <- convertFromExcelRef(table_range_refs)[[1]]
```

Then we can use the `removeTable` function to remove the existing named Excel table:

```
removeTable(wb = wb,
            sheet = 'input_data',
            table = 'example_table')
```

Now we can use `writeDataTable` to write the `iris` data frame (which comes with R) back into our workbook object in R:

```
writeDataTable(
  wb = wb,
  sheet = 'input_data',
  x = iris,
  startCol = table_col_num,
  startRow = table_row_num,
  tableStyle = "TableStyleLight9",
  tableName = 'example_table'
)
```

At this point we could save the workbook and our table would be updated. However, it's a good idea to save some metadata in the workbook to let others know exactly when the data was refreshed. We can do this with the `writeData` function, then save the workbook to a file and overwrite the original file. In this example, we'll put the metadata text in cell B:5, then save the workbook back to a file, overwriting the original:

```
writeData(
  wb = wb,
  sheet = 'input_data',
  x = paste('example_table data refreshed on:', Sys.time()),
  startCol = 2,
  startRow = 5
)

# then save the workbook
saveWorkbook(wb = wb,
```

```
file = "data/excel_table_data.xlsx",
overwrite = TRUE)
```

The resulting Excel sheet is shown in [Figure 4-2](#).

A screenshot of a Microsoft Excel spreadsheet. The ribbon at the top shows tabs for Home, Insert, Draw, Page Layout, Formulas, Data, Review, View, and Table. The Table tab is selected. A dropdown menu under Table Name shows 'example_table'. On the far left, there's a vertical column of row numbers from 1 to 15. The first five rows contain metadata: row 1 has 'example_table'; row 2 has 'data refreshed on: 2018-11-04 13:29:11'; rows 3 through 5 are empty. Row 10 contains the column headers: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species. Rows 11 through 15 contain data points for the 'setosa' species of the Iris dataset. The data is presented in a grid with horizontal and vertical grid lines.

	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5	example_table data refreshed on: 2018-11-04 13:29:11									
6										
7										
8										
9										
10	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species					
11	5.1	3.5	1.4	0.2	setosa					
12	4.9	3	1.4	0.2	setosa					
13	4.7	3.2	1.3	0.2	setosa					
14	4.6	3.1	1.5	0.2	setosa					
15	5	3.6	1.4	0.2	setosa					
-	--	--	--	--	--					

Figure 4-2. Excel table and metadata text

See Also

You can see the vignette for `openxlsx` by installing `openxlsx` and running `vignette('Introduction', package='openxlsx')`.

The [readxl package](#) is part of the tidyverse and provides fast, simple reading of Excel files (discussed in [Recipe 4.11](#)).

The [writexl package](#) is a fast and lightweight (no dependencies) package for writing Excel files.

4.13 Reading Data from a SAS File

Problem

You want to read a Statistical Analysis Software (SAS) dataset into an R data frame.

Solution

The `sas7bdat` package supports reading `.sas7bdat` files into R:

```
library(haven)  
  
sas_movie_data <- read_sas("data/movies.sas7bdat")
```

Discussion

SAS V7 and beyond all support the *.sas7bdat* file format. The `read_sas` function in `haven` supports reading the *.sas7bdat* file format, including variable labels. If your SAS file has variable labels, when they are imported into R they will be stored in the `label` attributes of the data frame. These labels will not be printed by default. You can see the labels by opening the data frame in RStudio, or by calling the `attributes` Base R function on each column:

```
sapply(sas_movie_data, attributes)  
#> $Movie  
#> $Movie$label  
#> [1] "Movie"  
#>  
#>  
#> $Type  
#> $Type$label  
#> [1] "Type"  
#>  
#>  
#> $Rating  
#> $Rating$label  
#> [1] "Rating"  
#>  
#>  
#> $Year  
#> $Year$label  
#> [1] "Year"  
#>  
#>  
#> $Domestic__  
#> $Domestic__$label  
#> [1] "Domestic $"  
#>  
#> $Domestic__$format.sas  
#> [1] "F"  
#>  
#>  
#> $Worldwide__  
#> $Worldwide__$label  
#> [1] "Worldwide $"  
#>  
#> $Worldwide__$format.sas  
#> [1] "F"  
#>  
#>  
#> $Director
```

```
#> $Director$label  
#> [1] "Director"
```

See Also

The `sas7bdat` package is much slower on large files than `haven`, but it has more elaborate support for file attributes. If the SAS metadata is important to you, then you should investigate `sas7bdat::read.sas7bdat`.

4.14 Reading Data from HTML Tables

Problem

You want to read data from an HTML table on the web.

Solution

Use the `read_html` and `html_table` functions in the `rvest` package. To read all tables on the page, do the following:

```
library(rvest)  
library(tidyverse)  
  
all_tables <-  
  read_html("url") %>%  
  html_table(fill = TRUE, header = TRUE)
```

Note that `rvest` is installed when you run `install.packages('tidyverse')`, but it is not a core tidyverse package. So, you must explicitly load the package.

Discussion

Web pages can contain several HTML tables. Calling `read_html(url)` and then piping that to `html_table` reads all tables on the page and returns them in a list. This can be useful for exploring a page, but it's annoying if you want just one specific table. In that case, use `extract2(n)` to select the *n*th table.

For example, here we extract all tables from a Wikipedia article:

```
library(rvest)  
  
all_tables <-  
  read_html("https://en.wikipedia.org/wiki/Aviation_accidents_and_incidents") %>%  
  html_table(fill = TRUE, header = TRUE)
```

`read_html` puts all the tables from the HTML document into the output list. To pull a single table from that list, you can use the function `extract2` from the `magrittr` package:

```

out_table <-
  all_tables %>%
  magrittr::extract2(2)

head(out_table)
#>   Year Deaths[53] # of incidents[54]
#> 1 2018     1,040          113[55]
#> 2 2017      399           101
#> 3 2016      629           102
#> 4 2015      898           123
#> 5 2014     1,328          122
#> 6 2013      459           138

```

Two common parameters for the `html_table` function are `fill=TRUE`, which fills in missing values with `NA`, and `header=TRUE`, which indicates that the first row contains the header names.

The following example loads all tables from the Wikipedia page entitled “World population”:

```

url <- 'http://en.wikipedia.org/wiki/World_population'
tbls <- read_html(url) %>%
  html_table(fill = TRUE, header = TRUE)

```

As it turns out, that page contains 23 tables (or things that `html_table` thinks might be tables):

```

length(tbls)
#> [1] 23

```

In this example we care only about the sixth table (which lists the largest populations by country), so we can either access that element using brackets—`tbls[[6]]`—or we can pipe it into the `extract2` function from the `magrittr` package:

```

library(magrittr)
tbl <- tbls %>%
  extract2(6)

head(tbl, 2)
#>   Rank Country / Territory   Population           Date % of world population
#> 1    1       China[note 4] 1,397,280,000 May 11, 2019            18.1%
#> 2    2           India 1,347,050,000 May 11, 2019            17.5%
#>   Source
#> 1  [84]
#> 2  [85]

```

The `extract2` function is a “pipe-friendly” version of the R `[[i]]` syntax: it pulls out a single list element from a list. The `extract` function is analogous to `[i]`, which returns element i from the original list into a list of length 1.

In that table, columns 2 and 3 contain the country name and population, respectively:

```
tbl[, c(2, 3)]
#>   Country / Territory   Population
#> 1      China[note 4] 1,397,280,000
#> 2          India 1,347,050,000
#> 3  United States 329,181,000
#> 4      Indonesia 265,015,300
#> 5       Pakistan 212,742,631
#> 6        Brazil 209,889,000
#> 7       Nigeria 188,500,000
#> 8     Bangladesh 166,532,000
#> 9      Russia[note 5] 146,877,088
#> 10       Japan 126,440,000
```

Right away, we can see problems with the data: China and Russia have [note 4] and [note 5] appended to their names. On the Wikipedia website those were footnote references, but now they're just bits of unwanted text. Adding insult to injury, the population numbers have embedded commas, so you cannot easily convert them to raw numbers. All these problems can be solved by some string processing, but each problem adds at least one more step to the process.

This illustrates the main obstacle to reading HTML tables. HTML was designed for presenting information to people, not to computers. When you “scrape” information off an HTML page, you get stuff that’s useful to people but annoying to computers. If you ever have a choice, choose instead a computer-oriented data representation such as XML, JSON, or CSV.



The `read_html(url)` and `html_table` functions are part of the `rvest` package, which (by necessity) is large and complex. Any time you pull data from a site designed for human readers, not machines, expect that you will have to do post-processing to clean up the bits and pieces the machine leaves messy.

See Also

See [Recipe 3.10](#) for downloading and installing packages such as the `rvest` package.

4.15 Reading Files with a Complex Structure

Problem

You are reading data from a file that has a complex or irregular structure.

Solution

Use the `readLines` function to read individual lines; then process them as strings to extract data items.

Alternatively, use the `scan` function to read individual tokens and use the argument `what` to describe the stream of tokens in your file. The function can convert tokens into data and then assemble the data into records.

Discussion

Life would be simple and beautiful if all our data files were organized into neat tables with cleanly delimited data. We could read those files using one of the functions in the `readr` package and get on with living.

Unfortunately, we don't live in a land of rainbows and unicorn kisses.

You will eventually encounter a funky file format, and your job is to read the file's contents into R.

The `read.table` and `read.csv` functions are file-oriented and probably won't help. However, the `readLines` and `scan` functions are useful here because they let you process the individual lines and even tokens of the file.

The `readLines` function is pretty simple. It reads lines from a file and returns them as a list of character strings:

```
lines <- readLines("input.txt")
```

You can limit the number of lines by using the `n` parameter, which gives the maximum number of lines to be read:

```
lines <- readLines("input.txt", n = 10)      # Read 10 lines and stop
```

The `scan` function is much richer. It reads one token at a time and handles it according to your instructions. The first argument is either a filename or a connection. The second argument is called `what`, and it describes the tokens that `scan` should expect in the input file. The description is cryptic but quite clever:

```
what=numeric(0)
    Interprets the next token as a number

what=integer(0)
    Interprets the next token as an integer

what=complex(0)
    Interprets the next token as a complex number

what=character(0)
    Interprets the next token as a character string

what=logical(0)
    Interprets the next token as a logical value
```

The `scan` function will apply the given pattern repeatedly until all data is read.

Suppose your file is simply a sequence of numbers, like this:

```
2355.09 2246.73 1738.74 1841.01 2027.85
```

Use `what=numeric(0)` to say, “My file is a sequence of tokens, each of which is a number”:

```
singles <- scan("./data/singles.txt", what = numeric(0))
singles
#> [1] 2355.09 2246.73 1738.74 1841.01 2027.85
```

A key feature of `scan` is that the `what` can be a list containing several token types. The `scan` function will assume your file is a repeating sequence of those types. Suppose your file contains triplets of data, like this:

```
15-Oct-87 2439.78 2345.63 16-Oct-87 2396.21 2207.73
19-Oct-87 2164.16 1677.55 20-Oct-87 2067.47 1616.21
21-Oct-87 2081.07 1951.76
```

Use a list to tell `scan` that it should expect a repeating, three-token sequence:

```
triples <-
  scan("./data/triples.txt",
    what = list(character(0), numeric(0), numeric(0)))
triples
#> [[1]]
#> [1] "15-Oct-87" "16-Oct-87" "19-Oct-87" "20-Oct-87" "21-Oct-87"
#>
#> [[2]]
#> [1] 2439.78 2396.21 2164.16 2067.47 2081.07
#>
#> [[3]]
#> [1] 2345.63 2207.73 1677.55 1616.21 1951.76
```

Give names to the list elements, and `scan` will assign those names to the data:

```
triples <- scan("./data/triples.txt",
  what = list(
    date = character(0),
    high = numeric(0),
    low = numeric(0)
  ))
triples
#> $date
#> [1] "15-Oct-87" "16-Oct-87" "19-Oct-87" "20-Oct-87" "21-Oct-87"
#>
#> $high
#> [1] 2439.78 2396.21 2164.16 2067.47 2081.07
#>
#> $low
#> [1] 2345.63 2207.73 1677.55 1616.21 1951.76
```

This can easily be turned into a data frame with the `data.frame` command:

```
df_triples <- data.frame(triples)
df_triples
#>      date    high    low
#> 1 15-Oct-87 2439.78 2345.63
#> 2 16-Oct-87 2396.21 2207.73
#> 3 19-Oct-87 2164.16 1677.55
#> 4 20-Oct-87 2067.47 1616.21
#> 5 21-Oct-87 2081.07 1951.76
```

The `scan` function has many bells and whistles, but the following are especially useful:

`n=number`

Stop after reading this many tokens. (Default: stop at end of file.)

`nlines=number`

Stop after reading this many input lines. (Default: stop at end of file.)

`skip=number`

Number of input lines to skip before reading data.

`na.strings=list`

A list of strings to be interpreted as NA.

An Example

Let's use this recipe to read a dataset from StatLib, the repository of statistical data and software maintained by Carnegie Mellon University. Jeff Witmer contributed a dataset called `wseries` that shows the pattern of wins and losses for every World Series since 1903. The dataset is stored in an ASCII file with 35 lines of comments followed by 23 lines of data. The data itself looks like this:

1903	LWLlwwWW	1927	wwWW	1950	wwWW	1973	WLwllWW
1905	wLwWW	1928	WWww	1951	LWlwWW	1974	wLwWW
1906	wLwLwW	1929	wwLWW	1952	lwLWLww	1975	lwWLwLw
1907	WWww	1930	WWllwW	1953	WWllww	1976	WWww
1908	wwLww	1931	LwwlwLW	1954	WWww	1977	WLwwlW
.							
.	(etc.)						
.							

The data is encoded as follows: L = loss at home, l = loss on the road, W = win at home, w = win on the road. The data appears in column order, not row order, which complicates our lives a bit.

Here is the R code for reading the raw data:

```
# Read the wseries dataset:
#   - Skip the first 35 lines
#   - Then read 23 lines of data
#   - The data occurs in pairs: a year and a pattern (char string)
```

```

#
world.series <- scan(
  "http://lib.stat.cmu.edu/datasets/wseries",
  skip = 35,
  nlines = 23,
  what = list(year = integer(0),
              pattern = character(0)),
  )

```

The `scan` function returns a list, so we get a list with two elements: `year` and `pattern`. The function reads from left to right, but the dataset is organized by columns and so the years appear in a strange order:

```

world.series$year
#> [1] 1903 1927 1950 1973 1905 1928 1951 1974 1906 1929 1952 1975 1907 1930
#> [15] 1953 1976 1908 1931 1954 1977 1909 1932 1955 1978 1910 1933 1956 1979
#> [29] 1911 1934 1957 1980 1912 1935 1958 1981 1913 1936 1959 1982 1914 1937
#> [43] 1960 1983 1915 1938 1961 1984 1916 1939 1962 1985 1917 1940 1963 1986
#> [57] 1918 1941 1964 1987 1919 1942 1965 1988 1920 1943 1966 1989 1921 1944
#> [71] 1967 1990 1922 1945 1968 1991 1923 1946 1969 1992 1924 1947 1970 1993
#> [85] 1925 1948 1971 1926 1949 1972

```

We can fix that by sorting the list elements according to year:

```

perm <- order(world.series$year)
world.series <- list(year = world.series$year[perm],
                      pattern = world.series$pattern[perm])

```

Now the data appears in chronological order:

```

world.series$year
#> [1] 1903 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917
#> [15] 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931
#> [29] 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945
#> [43] 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959
#> [57] 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973
#> [71] 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987
#> [85] 1988 1989 1990 1991 1992 1993

world.series$pattern
#> [1] "LWLlwww" "wlwww" "wLwlww" "Wwww" "wWLww" "WLwlwlw"
#> [7] "Wwwwlw" "lWwlw" "wLwlwlw" "wlwww" "wwnlw" "lwnlw"
#> [13] "WWlwlw" "WWlllw" "wlwlwlw" "WWlwllw" "wllwwww" "Lwlwlwlw"
#> [19] "WWww" "Lwlwlww" "Lwlwlwlw" "Lwlwlww" "lwlwllw" "wwnlw"
#> [25] "WWww" "wwlw" "WWlllw" "Lwlwlw" "wlww" "WWlww"
#> [31] "wlwlllw" "Lwwwlw" "lwnwlw" "WWlllw" "wwnlw" "WWww"
#> [37] "Lwlwlww" "WLwww" "Lwww" "WLlw" "Lwlwllw" "Lwlwllw"
#> [43] "Lwlwlww" "WWlllw" "lwlwlw" "WLwww" "wwnlw" "Lwlwllw"
#> [49] "lwlwlww" "WWlllw" "WWlllw" "lwlwlw" "llwwwlw" "lwlwlwlw"
#> [55] "llwlwlww" "lwlwlw" "WLlwlw" "WLwww" "wlwlwlw" "wwnlw"
#> [61] "WLlwlw" "llwwwlw" "wwnlw" "wlwlwlw" "lwlwllw" "lwlwllw"
#> [67] "wwnlw" "llwwwlw" "wwnlw" "WLwlwlw" "wlwllw" "lwlwlwlw"
#> [73] "WWww" "WLwwlw" "llwwwlw" "llwwwlw" "wlwllw" "llwwwlw"

```

```
#> [79] "LWwllWW"  "LWwww"     "wlWWW"      "LLwlwWW"    "LLwwlWW"    "WWlllWW"  
#> [85] "WwlWW"     "Wwww"       "Wwww"       "WWlllWW"    "lwWWlW"     "WLwwlW"
```

4.16 Reading from MySQL Databases

Problem

You want access to data stored in a MySQL database.

Solution

Follow these steps:

1. Install the RMySQL package on your computer and add a user and password.
2. Open a database connection using the DBI::dbConnect function.
3. Use dbGetQuery to initiate a SELECT and return the result sets.
4. Use dbDisconnect to terminate the database connection when you are done.

Discussion

This recipe requires that the RMySQL package be installed on your computer. That package requires, in turn, the MySQL client software. If that software is not already installed and configured on your system, consult the MySQL documentation or your system administrator.

Use the dbConnect function to establish a connection to the MySQL database. It returns a connection object that is used in subsequent calls to RMySQL functions:

```
library(RMySQL)  
  
con <- dbConnect(  
  drv = RMySQL::MySQL(),  
  dbname = "your_db_name",  
  host = "your.host.com",  
  username = "userid",  
  password = "pwd")
```

The `username`, `password`, and `host` parameters are the same parameters used for accessing MySQL through the `mysql` client program. The example given here shows them hardcoded into the `dbConnect` call, but actually that is an ill-advised practice. It puts your password in a plain-text document, creating a security problem. It also creates a major headache whenever your password or host changes, requiring you to hunt down the hardcoded values. We strongly recommend using the security mechanism of MySQL instead. Version 8 of MySQL introduces even more advanced secu-

riety options, but currently these have not been built into the RMySQL client. So, we recommend you use MySQL native passwords by setting `default-authentication-plugin=mysql_native_password` in your MySQL configuration file, which is `$HOME/.my.cnf` on Unix and `C:\my.cnf` on Windows. We use `loose-local-infile=1` to ensure that we have permissions to write to the database. Make sure the file is unreadable by anyone except you. The file is delimited into sections with markers such as `[mysqld]` and `[client]`. Put connection parameters into the `[client]` section, so that your config file will contain something like this:

```
[mysqld]
default-authentication-plugin=mysql_native_password
loose-local-infile=1

[client]
loose-local-infile=1
user="jdl"
password="password"
host=127.0.0.1
port=3306
```

Once the parameters are defined in the config file, you no longer need to supply them in the `dbConnect` call, which then becomes much simpler:

```
con <- dbConnect(
  drv = RMySQL::MySQL(),
  dbname = "your_db_name")
```

Use the `dbGetQuery` function to submit your SQL to the database and read the result sets. Doing so requires an open database connection:

```
sql <- "SELECT * from SurveyResults WHERE City = 'Chicago'"
rows <- dbGetQuery(con, sql)
```

You are not restricted to `SELECT` statements. Any SQL that generates a result set is OK. It is common to use `CALL` statements, for example, if your SQL is encapsulated in stored procedures and those stored procedures contain embedded `SELECT` statements.

Using `dbGetQuery` is convenient because it packages the result set into a data frame and returns the data frame. This is the perfect representation of a SQL result set. The result set is a tabular data structure of rows and columns, and so is a data frame. The result set's columns have names given by the SQL `SELECT` statement, and R uses them for naming the columns of the data frame.

Call `dbGetQuery` repeatedly to perform multiple queries. When you are done, close the database connection using `dbDisconnect`:

```
dbDisconnect(con)
```

Here is a complete session that reads and prints three rows from a database of stock prices. The query selects the price of IBM stock for the last three days of 2008. It

assumes that the `username`, `password`, `dbname`, and `host` parameters are defined in the `my.cnf` file:

```
con <- dbConnect(RMySQL::MySQL())
sql <- paste(
  "select * from DailyBar where Symbol = 'IBM'",
  "and Day between '2008-12-29' and '2008-12-31'"
)
rows <- dbGetQuery(con, sql)

dbDisconnect(con)
print(rows)

##   Symbol      Day     Next OpenPx HighPx LowPx ClosePx AdjClosePx
## 1   IBM 2008-12-29 2008-12-30  81.72  81.72 79.68   81.25    81.25
## 2   IBM 2008-12-30 2008-12-31  81.83  83.64 81.52   83.55    83.55
## 3   IBM 2008-12-31 2009-01-02  83.50  85.00 83.50   84.16    84.16
##   HistClosePx  Volume OpenInt
## 1       81.25 6062600      NA
## 2       83.55 5774400      NA
## 3       84.16 6667700      NA
```

See Also

See [Recipe 3.10](#) and the documentation for `RMySQL`, which contains more details about configuring and using the package.

See [Recipe 4.17](#) for information about how to get data from a SQL database without writing any SQL.

R can read from several other RDBMSs, including Oracle, Sybase, PostgreSQL, and SQLite. For more information, see the R Data Import/Export guide, which is supplied with the base distribution ([Recipe 1.7](#)) and is also available on [CRAN](#).

4.17 Accessing a Database with `dbplyr`

Problem

You want to access a database, but you'd rather not write SQL code in order to manipulate data and return results to R.

Solution

In addition to being a grammar of data manipulation, the tidyverse package `dplyr` can, in connection with the `dbplyr` package, turn `dplyr` commands into SQL for you.

Let's set up an example database using `RSQlite`. Then we'll connect to it and use `dplyr` and the `dbplyr` backend to extract data.

We'll first set up the example table by loading the `msleep` example data into an in-memory SQLite database:

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
sleep_db <- copy_to(con, msleep, "sleep")
```

Now that we have a table in our database, we can create a reference to it from R:

```
sleep_table <- tbl(con, "sleep")
```

The `sleep_table` object is a type of pointer or alias to the table on the database. However, `dplyr` will treat it like a regular tidyverse tibble or data frame, so you can operate on it using `dplyr` and other R commands. Let's select all animals from the data who sleep less than three hours:

```
little_sleep <- sleep_table %>%
  select(name, genus, order, sleep_total) %>%
  filter(sleep_total < 3)
```

The `dbplyr` backend does not go fetch the data when we do the preceding commands. But it does build the query and get ready. To see the query built by `dplyr`, you can use `show_query`:

```
show_query(little_sleep)
#> <SQL>
#> SELECT *
#> FROM (SELECT `name`, `genus`, `order`, `sleep_total` 
#> FROM `sleep`)
#> WHERE (`sleep_total` < 3.0)
```

To bring the data back to your local machine, use `collect`:

```
local_little_sleep <- collect(little_sleep)
local_little_sleep
#> # A tibble: 3 x 4
#>   name      genus     order    sleep_total
#>   <chr>     <chr>     <chr>        <dbl>
#> 1 Horse     Equus    Perissodactyla     2.9
#> 2 Giraffe   Giraffa   Artiodactyla      1.9
#> 3 Pilot whale Globicephalus Cetacea    2.7
```

Discussion

When you use `dplyr` to access SQL databases by writing only `dplyr` commands, you can be more productive by not having to switch from one language to another and back. The alternative is to have large chunks of SQL code stored as text strings in the middle of an R script, or have the SQL in separate files that are read in by R.

By allowing `dplyr` to transparently create the SQL in the background, you are freed from having to maintain separate SQL code to extract data.

The `dbplyr` package uses `DBI` to connect to your database, so you'll need a `DBI` backend package for whichever database you want to access.

Some commonly used `DBI` backend packages are:

`odbc`

Uses the Open Database Connectivity (ODBC) protocol to connect to many different databases. This is typically the best choice when you are connecting to Microsoft SQL Server. ODBC is typically straightforward on Windows machines but may require some considerable effort to get working in Linux or macOS.

`RPostgreSQL`

For connecting to Postgres and Redshift.

`RMySQL`

For MySQL and MariaDB.

`RSQlite`

For connecting to SQLite databases on disk or in memory.

`bigrquery`

For connections to Google's BigQuery.



Each `DBI` backend package discussed here is listed on CRAN and can be installed with the typical `install.packages('package name')` command.

See Also

For more information about connecting the databases with R and RStudio, see <https://db.rstudio.com/>.

For more detail on SQL translation in `dbplyr`, see the `sql-translation` vignette at `vignette("sql-translation")` or <http://bit.ly/2wVCOKe>.

4.18 Saving and Transporting Objects

Problem

You want to store one or more R objects in a file for later use, or you want to copy an R object from one machine to another.

Solution

Write the objects to a file using the `save` function:

```
save(tbl, t, file = "myData.RData")
```

Read them back using the `load` function, either on your computer or on any platform that supports R:

```
load("myData.RData")
```

The `save` function writes binary data. To save in an ASCII format, use `dput` or `dump` instead:

```
dput(tbl, file = "myData.txt")
dump("tbl", file = "myData.txt")      # Note quotes around variable name
```

Discussion

Suppose you've found yourself with a large, complicated data object that you want to load into other workspaces, or you want to move R objects between a Linux box and a Windows box. The `load` and `save` functions let you do all this: `save` will store the object in a file that is portable across machines, and `load` can read those files.

When you run `load`, it does not return your data per se; rather, it creates variables in your workspace, loads your data into those variables, and then returns the names of the variables (in a vector). The first time you run `load`, you might be tempted to do this:

```
myData <- load("myData.RData")      # Achtung! Might not do what you think
```

Let's look at what `myData` is:

```
myData
#> [1] "tbl" "t"
str(myData)
#> chr [1:2] "tbl" "t"
```

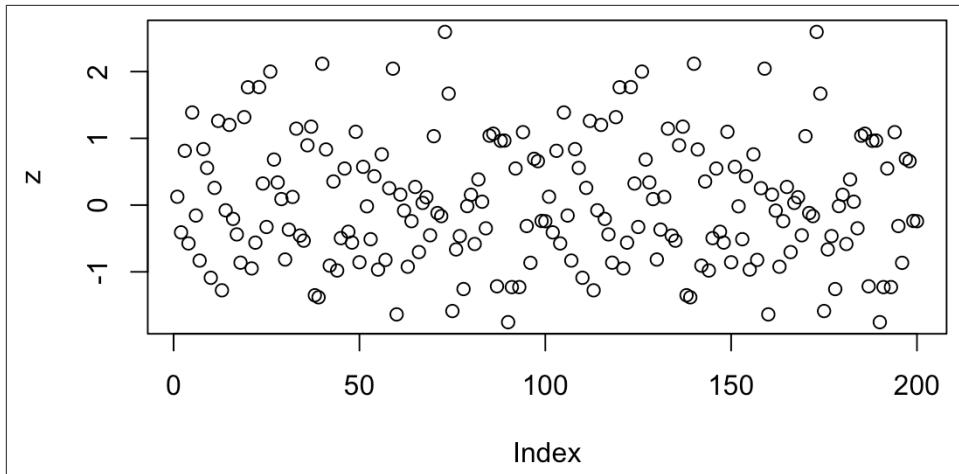
This might be puzzling, because `myData` will not contain your data at all. This can be perplexing and frustrating the first time you encounter it.

There are a few other things to keep in mind, too. First, the `save` function writes in a binary format to keep the file small. Sometimes you want an ASCII format instead. When you submit a question to a mailing list or to Stack Overflow, for example, including an ASCII dump of the data lets others re-create your problem. In such cases use `dput` or `dump`, which write an ASCII representation.

You must also be careful when you save and load objects created by a particular R package. When you load the objects, R does not automatically load the required packages, too, so it will not “understand” the object unless you previously loaded the package yourself. For instance, suppose we have an object called `z` created by the `zoo` package, and we save the object in a file called `z.RData`. The following sequence of functions will create some confusion:

```
load("./data/z.RData") # Create and populate the z variable  
plot(z) # Does not plot as expected: zoo pkg not loaded
```

The plot in [Figure 4-3](#) shows the resulting plot, which is just points.

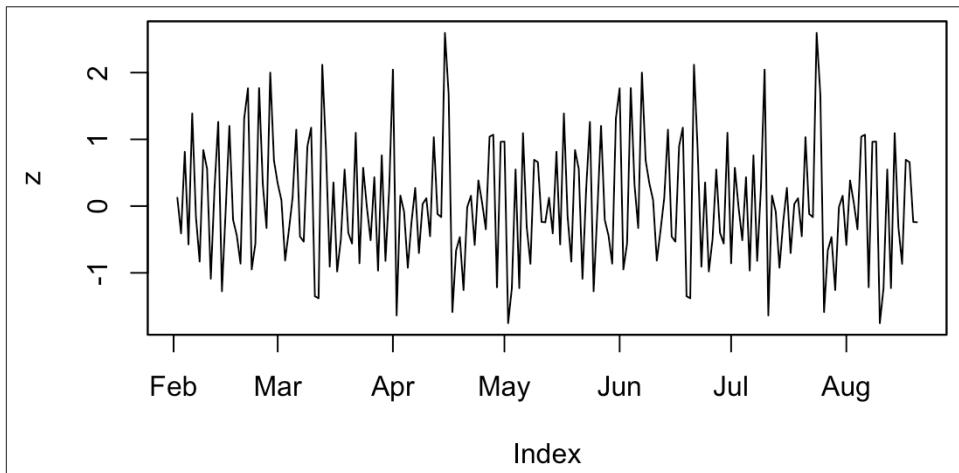


[Figure 4-3](#). Plot without *zoo* loaded

We should have loaded the *zoo* package *before* printing or plotting any *zoo* objects, like this:

```
library(zoo) # Load the zoo package into memory  
load("./data/z.RData") # Create and populate the z variable  
plot(z) # Ahhh. Now plotting works correctly
```

You can see the resulting plot in [Figure 4-4](#).



[Figure 4-4](#). Plotting with *zoo*

See Also

If you are just saving and loading a single data frame or other R object, you should consider `write_rds` and `read_rds`. These functions don't have "side effects" like `load`.

CHAPTER 5

Data Structures

You can get pretty far in R just using vectors. That's what [Chapter 2](#) is all about. This chapter moves beyond vectors to recipes for matrices, lists, factors, data frames, and tibbles (which are a special kind of data frame). If you have preconceptions about data structures, we suggest you put them aside. R does data structures differently than many other languages. Before we get to the recipes in this chapter, we'll take a quick look at different data structures in R.

If you want to study the technical aspects of R's data structures, we suggest reading [R in a Nutshell](#) and the [R Language Definition](#). The notes here are more informal. These are things we wish we'd known when we started using R.

Vectors

Here are some key properties of vectors:

Vectors are homogeneous.

All elements of a vector must have the same type or, in R terminology, the same *mode*.

Vectors can be indexed by position.

So `v[2]` refers to the second element of `v`.

Vectors can be indexed by multiple positions, returning a subvector.

So `v[c(2,3)]` is a subvector of `v` that consists of the second and third elements.

Vector elements can have names.

Vectors have a `names` property, the same length as the vector itself, that gives names to the elements:

```
v <- c(10, 20, 30)
names(v) <- c("Moe", "Larry", "Curly")
```

```
print(v)
#> Moe Larry Curly
#> 10 20 30
```

If vector elements have names, then you can select them by name.

Continuing the previous example:

```
v[["Larry"]]
#> [1] 20
```

Lists

Here are some key properties of lists:

Lists are heterogeneous.

Lists can contain elements of different types—in R terminology, list elements may have different modes. Lists can even contain other structured objects, such as lists and data frames; this allows you to create recursive data structures.

Lists can be indexed by position.

So `lst[[2]]` refers to the second element of `lst`. Note the double square brackets. Double brackets means that R will return the element as whatever type of element it is.

Lists let you extract sublists.

So `lst[c(2,3)]` is a sublist of `lst` that consists of the second and third elements. Note the single square brackets. Single brackets means that R will return the items in a list. If you pull a single element with single brackets, like `lst[2]`, R will return a list of length 1 with the first item being the desired item.

List elements can have names.

Both `lst[["Moe"]]` and `lst$Moe` refer to the element named “Moe.”

Since lists are heterogeneous and since their elements can be retrieved by name, a list is like a dictionary or hash or lookup table in other programming languages (discussed in [Recipe 5.9](#)).

What’s surprising (and cool) is that in R, unlike most of those other programming languages, lists can also be indexed by position.

Mode: Physical Type

In R, every object has a mode, which indicates how it is stored in memory: as a number, as a character string, as a list of pointers to other objects, as a function, and so forth (see [Table 5-1](#)).

Table 5-1. R object-mode mapping

Object	Example	Mode
Number	3.1415	Numeric
Vector of numbers	c(2.7.182, 3.1415)	Numeric
Character string	"Moe"	Character
Vector of character strings	c("Moe", "Larry", "Curly")	Character
Factor	factor(c("NY", "CA", "IL"))	Numeric
List	list("Moe", "Larry", "Curly")	List
Data frame	data.frame(x=1:3, y=c("NY", "CA", "IL"))	List
Function	print	Function

The `mode` function gives us this information:

```
mode(3.1415)                      # Mode of a number
#> [1] "numeric"
mode(c(2.7182, 3.1415))          # Mode of a vector of numbers
#> [1] "numeric"
mode("Moe")                        # Mode of a character string
#> [1] "character"
mode(list("Moe", "Larry", "Curly")) # Mode of a list
#> [1] "list"
```

A critical difference between vectors and lists can be summed up this way:

- In a vector, all elements must have the same mode.
- In a list, the elements can have different modes.

Class: Abstract Type

In R, every object also has a class, which defines its abstract type. The terminology is borrowed from object-oriented programming. A single number could represent many different things: a distance, a point in time, or a weight, for example. All those objects have a mode of "numeric" because they are stored as a number, but they could have different classes to indicate their interpretation.

For example, a `Date` object consists of a single number:

```
d <- as.Date("2010-03-15")
mode(d)
#> [1] "numeric"
length(d)
#> [1] 1
```

But it has a class of `Date`, telling us how to interpret that number—namely, as the number of days since January 1, 1970:

```
class(d)
#> [1] "Date"
```

R uses an object's class to decide how to process the object. For example, the generic function `print` has specialized versions (called *methods*) for printing objects according to their class: `data.frame`, `Date`, `lm`, and so forth. When you print an object, R calls the appropriate `print` function according to the object's class.

Scalars

The quirky thing about scalars is their relationship to vectors. In some software, scalars and vectors are two different things. In R, they are the same thing: a scalar is simply a vector that contains exactly one element. In this book we often use the term “scalar,” but that’s just shorthand for “vector with one element.”

Consider the built-in constant `pi`. It is a scalar:

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

Since a scalar is a one-element vector, you can use vector functions on `pi`:

```
length(pi)
#> [1] 1
```

You can index it. The first (and only) element is π , of course:

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

If you ask for the second element, there is none:

```
pi[2]
#> [1] NA
```

Matrices

In R, a matrix is just a vector that has dimensions. It may seem strange at first, but you can transform a vector into a matrix simply by giving it dimensions.

A vector has an attribute called `dim`, which is initially `NULL`, as shown here:

```
A <- 1:6
dim(A)
#> NULL
print(A)
#> [1] 1 2 3 4 5 6
```

We give dimensions to the vector when we set its `dim` attribute. Watch what happens when we set our vector dimensions to 2×3 and print it:

```
dim(A) <- c(2, 3)
print(A)
```

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

Voilà! The vector was reshaped into a 2×3 matrix.

A matrix can be created from a list, too. Like a vector, a list has a `dim` attribute, which is initially `NULL`:

```
B <- list(1, 2, 3, 4, 5, 6)
dim(B)
#> NULL
```

If we set the `dim` attribute, it gives the list a shape:

```
dim(B) <- c(2, 3)
print(B)
#>      [,1] [,2] [,3]
#> [1,] 1    3    5
#> [2,] 2    4    6
```

Voilà! We have turned this list into a 2×3 matrix.

Arrays

The discussion of matrices can be generalized to three-dimensional or even n -dimensional structures: just assign more dimensions to the underlying vector (or list). The following example creates a three-dimensional array with dimensions $2 \times 3 \times 2$:

```
D <- 1:12
dim(D) <- c(2, 3, 2)
print(D)
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
```

Note that R prints one “slice” of the structure at a time, since it’s not possible to print a three-dimensional structure on a two-dimensional medium.

It strikes us as very odd that we can turn a list into a matrix just by giving the list a `dim` attribute. But wait: it gets stranger.

Recall that a list can be heterogeneous (mixed modes). We can start with a heterogeneous list, give it dimensions, and thus create a heterogeneous matrix. This code snippet creates a matrix that is a mix of numeric and character data:

```
C <- list(1, 2, 3, "X", "Y", "Z")
dim(C) <- c(2, 3)
print(C)
#>      [,1] [,2] [,3]
#> [1,] 1     3     "Y"
#> [2,] 2     "X"  "Z"
```

To us, this is strange because we ordinarily assume a matrix is purely numeric, not mixed. R is not that restrictive.

The possibility of a heterogeneous matrix may seem powerful and strangely fascinating. However, it creates problems when you are doing normal, day-to-day stuff with matrices. For example, what happens when the matrix C (from the previous example) is used in matrix multiplication? What happens if it is converted to a data frame? The answer is that *odd things happen*.

In this book, we generally ignore the pathological case of a heterogeneous matrix. We assume you've got simple, vanilla matrices. Some recipes involving matrices may work oddly (or not at all) if your matrix contains mixed data. Converting such a matrix to a vector or data frame, for instance, can be problematic (see [Recipe 5.29](#)).

Factors

A factor looks like a character vector, but it has special properties. R keeps track of the unique values in a vector, and each unique value is called a *level* of the associated factor. R uses a compact representation for factors, which makes them efficient for storage in data frames. In other programming languages, a factor would be represented by a vector of enumerated values.

There are two key uses for factors:

Categorical variables

A factor can represent a categorical variable. Categorical variables are used in contingency tables, linear regression, analysis of variance (ANOVA), logistic regression, and many other areas.

Grouping

This is a technique for labeling or tagging your data items according to their group. See [Chapter 6](#).

Data Frames

A data frame is a powerful and flexible structure. Most serious R applications involve data frames. A data frame is intended to mimic a dataset, such as one you might encounter in SAS or SPSS, or a table in an SQL database.

A data frame is a tabular (rectangular) data structure, which means that it has rows and columns. It is not implemented by a matrix, however. Rather, a data frame is a list with the following characteristics:

- The elements of the list are vectors and/or factors.¹
- Those vectors and factors are the columns of the data frame.
- The vectors and factors must all have the same length; in other words, all columns must have the same height.
- The equal-height columns give a rectangular shape to the data frame.
- The columns must have names.

Because a data frame is both a list and a rectangular structure, R provides two different paradigms for accessing its contents:

- You can use list operators to extract columns from a data frame, such as `df[i]`, `df[[i]]`, or `df$name`.
- You can use matrix-like notation, such as `df[i,j]`, `df[i,]`, or `df[,j]`.

Your perception of a data frame likely depends on your background:

To a statistician

A data frame is a table of observations. Each row contains one observation. Each observation must contain the same variables. These variables are called columns, and you can refer to them by name. You can also refer to the contents by row number and column number, just as with a matrix.

To a SQL programmer

A data frame is a table. The table resides entirely in memory, but you can save it to a flat file and restore it later. You needn't declare the column types because R figures that out for you.

¹ A data frame can be built from a mixture of vectors, factors, and matrices. The columns of the matrices become columns in the data frame. The number of *rows* in each matrix must match the *length* of the vectors and factors. In other words, all elements of a data frame must have the same *height*.

To an Excel user

A data frame is like a worksheet, or perhaps a range within a worksheet. It is more restrictive, however, in that each column has a type.

To an SAS user

A data frame is like an SAS dataset for which all the data resides in memory. R can read and write the data frame on disk, but the data frame must be in memory while R is processing it.

To an R programmer

A data frame is a hybrid data structure, part matrix and part list. A column can contain numbers, character strings, or factors, but not a mix of them. You can index the data frame just like you index a matrix. The data frame is also a list, where the list elements are the columns, so you can access columns by using list operators.

To a computer scientist

A data frame is a rectangular data structure. The columns are typed, and each column must contain numeric values, character strings, or factors. Columns must have labels; rows may have labels. The table can be indexed by position, column name, and/or row name. It can also be accessed by list operators, in which case R treats the data frame as a list whose elements are the columns of the data frame.

To a corporate executive

You can put names and numbers into a data frame. A data frame is like a little database. Your staff will enjoy using data frames.

Tibbles

A *tibble* is a modern reimagining of the data frame, introduced by Hadley Wickham in the `tibble` package, which is a core package in the tidyverse. Most of the common functions you would use with data frames also work with tibbles. However, tibbles typically do less than data frames and complain more. This idea of complaining and doing less may remind you of your least favorite coworker; however, we think tibbles will be one of your favorite data structures. Doing less and complaining more can be a feature, not a bug.

Unlike data frames, tibbles:

- Do not give you row numbers by default.
- Do not give you strange, unexpected column names.
- Don't coerce your data into factors (unless you explicitly ask for that).
- Recycle vectors of length 1 but not other lengths.

In addition to basic data frame functionality, tibbles:

- Print only the top four rows and a bit of metadata by default.
- Always return a tibble when subsetting.
- Never do partial matching: if you want a column from a tibble, you have to ask for it using its full name.
- Complain more by giving you more warnings and chatty messages to make sure you understand what the software is doing.

All these extras are designed to give you fewer surprises and help you make fewer mistakes.

5.1 Appending Data to a Vector

Problem

You want to append additional data items to a vector.

Solution

Use the vector constructor (`c`) to construct a vector with the additional data items:

```
v <- c(1, 2, 3)
newItems <- c(6, 7, 8)
c(v, newItems)
#> [1] 1 2 3 6 7 8
```

For a single item, you can also assign the new item to the next vector element. R will automatically extend the vector:

```
v <- c(1, 2, 3)
v[length(v) + 1] <- 42
v
#> [1] 1 2 3 42
```

Discussion

If you ask us about appending a data item to a vector, we will likely suggest that maybe you shouldn't.



R works best when you think about entire vectors, not single data items. Are you repeatedly appending items to a vector? If so, then you are probably working inside a loop. That's OK for small vectors, but for large vectors your program will run slowly. The memory management in R works poorly when you repeatedly extend a vector by one element. Try to replace that loop with vector-level operations. You'll write less code, and R will run much faster.

Nonetheless, one does occasionally need to append data to vectors. Our experiments show that the most efficient way of doing so is to create a new vector using the vector constructor (`c`) to join the old and new data. This works for appending single elements or multiple elements:

```
v <- c(1, 2, 3)
v <- c(v, 4) # Append a single value to v
v
#> [1] 1 2 3 4

w <- c(5, 6, 7, 8)
v <- c(v, w) # Append an entire vector to v
v
#> [1] 1 2 3 4 5 6 7 8
```

You can also append an item by assigning it to the position past the end of the vector, as shown in the Solution. In fact, R is very liberal about extending vectors. You can assign to any element and R will expand the vector to accommodate your request:

```
v <- c(1, 2, 3) # Create a vector of three elements
v[10] <- 10      # Assign to the 10th element
v
#> [1] 1 2 3 NA NA NA NA NA NA 10
```

Note that R did not complain about the out-of-bounds subscript. It just extended the vector to the needed length, filling it with `NA`.

R includes an `append` function that creates a new vector by appending items to an existing vector. However, our experiments show that this function runs more slowly than both the vector constructor and the element assignment.

5.2 Inserting Data into a Vector

Problem

You want to insert one or more data items into a vector.

Solution

Despite its name, the `append` function inserts data into a vector by using the `after` parameter, which gives the insertion point for the new item or items:

```
append(vec, newvalues, after = n)
```

Discussion

The new items will be inserted at the position given by `after`. This example inserts 99 into the middle of a sequence:

```
append(1:10, 99, after = 5)
#> [1] 1 2 3 4 5 99 6 7 8 9 10
```

The special value of `after=0` means insert the new items at the *head* of the vector:

```
append(1:10, 99, after = 0)
#> [1] 99 1 2 3 4 5 6 7 8 9 10
```

The comments in [Recipe 5.1](#) apply here, too. If you are inserting single items into a vector, you might be working at the element level when working at the vector level would be easier to code and faster to run.

5.3 Understanding the Recycling Rule

Problem

You want to understand the mysterious Recycling Rule that governs how R handles vectors of unequal length.

Discussion

When you do vector arithmetic, R performs element-by-element operations. That works well when both vectors have the same length: R pairs the elements of the vectors and applies the operation to those pairs.

But what happens when the vectors have unequal lengths?

In that case, R invokes the Recycling Rule. It processes the vector elements in pairs, starting at the first elements of both vectors. At a certain point, the shorter vector is exhausted while the longer vector still has unprocessed elements. R then returns to the beginning of the shorter vector, “recycling” its elements, while it continues taking elements from the longer vector until it completes the operation. It will recycle the shorter vector’s elements as often as necessary until the operation is complete.

It's useful to visualize the Recycling Rule. Here is a diagram of two vectors, `1:6` and `1:3`:

1:6	1:3
1	1
2	2
3	3
4	
5	
6	

Obviously, the `1:6` vector is longer than the `1:3` vector. If we try to add the vectors using `(1:6) + (1:3)`, it appears that `1:3` has too few elements. However, R recycles the elements of `1:3`, pairing the two vectors like this and producing a six-element vector:

1:6	1:3	<code>(1:6) + (1:3)</code>
1	1	2
2	2	4
3	3	6
4		5
5		7
6		9

Here is what you see in the R console:

```
(1:6) + (1:3)
#> [1] 2 4 6 5 7 9
```

It's not only vector operations that invoke the Recycling Rule; functions can, too. The `cbind` function can create column vectors, such as the following column vectors of `1:6` and `1:3`. The two columns have different heights, of course:

```
cbind(1:6)
```

```
cbind(1:3)
```

If we try binding these column vectors together into a two-column matrix, the lengths are mismatched. The `1:3` vector is too short, so `cbind` invokes the Recycling Rule and recycles the elements of `1:3`:

```
cbind(1:6, 1:3)
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    2
#> [3,]    3    3
#> [4,]    4    1
#> [5,]    5    2
#> [6,]    6    3
```

If the longer vector's length is not a multiple of the shorter vector's length, R gives a warning. That's good, since the operation is highly suspect and there is likely a bug in your logic:

```
(1:6) + (1:5) # Oops! 1:5 is one element too short
#> Warning in (1:6) + (1:5): longer object length is not a multiple of shorter
#> object length
#> [1] 2 4 6 8 10 7
```

Once you understand the Recycling Rule, you will realize that operations between a vector and a scalar are simply applications of that rule. In this example, the 10 is recycled repeatedly until the vector addition is complete:

```
(1:6) + 10
#> [1] 11 12 13 14 15 16
```

5.4 Creating a Factor (Categorical Variable)

Problem

You have a vector of character strings or integers. You want R to treat them as a factor, which is R's term for a categorical variable.

Solution

The `factor` function encodes your vector of discrete values into a factor:

```
f <- factor(v) # v can be a vector of strings or integers
```

If your vector contains only a subset of possible values and not the entire universe, then include a second argument that gives the possible levels of the factor:

```
f <- factor(v, levels)
```

Discussion

In R, each possible value of a categorical variable is called a *level*. A vector of levels is called a *factor*. Factors fit very cleanly into the vector orientation of R, and they are used in powerful ways for processing data and building statistical models.

Most of the time, converting your categorical data into a factor is a simple matter of calling the `factor` function, which identifies the distinct levels of the categorical data and packs them into a factor:

```
f <- factor(c("Win", "Win", "Lose", "Tie", "Win", "Lose"))
f
#> [1] Win Win Lose Tie Win Lose
#> Levels: Lose Tie Win
```

Notice that when we printed the factor, `f`, R did not put quotes around the values. They are levels, not strings. Also notice that when we printed the factor, R displayed the distinct levels below the factor.

If your vector contains only a subset of all the possible levels, then R will have an incomplete picture of the possible levels. Suppose you have a string-valued variable `wday` that gives the day of the week on which your data was observed:

```
wday <- c("Wed", "Thu", "Mon", "Wed", "Thu",
         "Thu", "Thu", "Tue", "Thu", "Tue")
f <- factor(wday)
f
#> [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
#> Levels: Mon Thu Tue Wed
```

R thinks that Monday, Thursday, Tuesday, and Wednesday are the only possible levels. Friday is not listed. Apparently, the lab staff never made observations on a Friday, so R does not know that Friday is a possible value. Hence, you need to list the possible levels of `wday` explicitly:

```
f <- factor(wday, levels=c("Mon", "Tue", "Wed", "Thu", "Fri"))
f
#> [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
#> Levels: Mon Tue Wed Thu Fri
```

Now R understands that `f` is a factor with five possible levels. It knows their correct order, too. It originally put Thursday before Tuesday because it assumes alphabetical order by default. The explicit `levels` argument defines the correct order.

In many situations it is not necessary to call `factor` explicitly. When an R function requires a factor, it usually converts your data to a factor automatically. The `table` function, for instance, works only on factors, so it routinely converts its inputs to factors without asking. You must explicitly create a factor variable when you want to specify the full set of levels or when you want to control the ordering of levels.

See Also

See [Recipe 12.5](#) to create a factor from continuous data.

5.5 Combining Multiple Vectors into One Vector and a Factor

Problem

You have several groups of data, with one vector for each group. You want to combine the vectors into one large vector and simultaneously create a parallel factor that identifies each value's original group.

Solution

Create a list that contains the vectors. Use the `stack` function to combine the list into a two-column data frame:

```
comb <- stack(list(v1 = v1, v2 = v2, v3 = v3)) # Combine 3 vectors
```

The data frame's columns are called `values` and `ind`. The first column contains the data, and the second column contains the parallel factor.

Discussion

Why in the world would you want to mash all your data into one big vector and a parallel factor? The reason is that many important statistical functions require the data in that format.

Suppose you survey freshmen, sophomores, and juniors regarding their confidence level (“What percentage of the time do you feel confident in school?”). Now you have three vectors, called `freshmen`, `sophomores`, and `juniors`. You want to perform an ANOVA of the differences between the groups. The `aov` function, requires one vector with the survey results as well as a parallel factor that identifies the group. You can combine the groups using the `stack` function:

```
freshmen <- c(1, 2, 1, 1, 5)
sophomores <- c(3, 2, 3, 3, 5)
juniors <- c(5, 3, 4, 3, 3)

comb <- stack(list(fresh = freshmen, soph = sophomores, jrs = juniors))
print(comb)
#>   values  ind
#> 1      1 fresh
#> 2      2 fresh
#> 3      1 fresh
#> 4      1 fresh
#> 5      5 fresh
#> 6      3 soph
#> 7      2 soph
#> 8      3 soph
#> 9      3 soph
#> 10     5 soph
#> 11     5    jrs
#> 12     3    jrs
#> 13     4    jrs
#> 14     3    jrs
#> 15     3    jrs
```

Now you can perform the ANOVA on the two columns:

```
aov(values ~ ind, data = comb)
```

When building the list we must provide tags for the list elements. (The tags are `fresh`, `soph`, and `jrs` in this example.) Those tags are required because `stack` uses them as the levels of the parallel factor.

5.6 Creating a List

Problem

You want to create and populate a list.

Solution

To create a list from individual data items, use the `list` function:

```
lst <- list(x, y, z)
```

Discussion

Lists can be quite simple, such as this list of three numbers:

```
lst <- list(0.5, 0.841, 0.977)
lst
#> [[1]]
#> [1] 0.5
#
#> [[2]]
#> [1] 0.841
#
#> [[3]]
#> [1] 0.977
```

When R prints the list, it identifies each list element by its position ([[1]], [[2]], [[3]]) and prints the element's value (e.g., [1] 0.5) under its position.

More usefully, lists can, unlike vectors, contain elements of different modes (types). Here is an extreme example of a mongrel created from a scalar, a character string, a vector, and a function:

```
lst <- list(3.14, "Moe", c(1, 1, 2, 3), mean)
lst
#> [[1]]
#> [1] 3.14
#
#> [[2]]
#> [1] "Moe"
#
#> [[3]]
#> [1] 1 1 2 3
#
#> [[4]]
```

```
#> function (x, ...)  
#> UseMethod("mean")  
#> <bytecode: 0x7ff04b0bc900>  
#> <environment: namespace:base>
```

You can also build a list by creating an empty list and populating it. Here is our “mongrel” example built in that way:

```
lst <- list()  
lst[[1]] <- 3.14  
lst[[2]] <- "Moe"  
lst[[3]] <- c(1, 1, 2, 3)  
lst[[4]] <- mean  
lst  
#> [[1]]  
#> [1] 3.14  
#>  
#> [[2]]  
#> [1] "Moe"  
#>  
#> [[3]]  
#> [1] 1 1 2 3  
#>  
#> [[4]]  
#> function (x, ...)  
#> UseMethod("mean")  
#> <bytecode: 0x7ff04b0bc900>  
#> <environment: namespace:base>
```

List elements can be named. The `list` function lets you supply a name for every element:

```
lst <- list(mid = 0.5, right = 0.841, far.right = 0.977)  
lst  
#> $mid  
#> [1] 0.5  
#>  
#> $right  
#> [1] 0.841  
#>  
#> $far.right  
#> [1] 0.977
```

See Also

See the introduction to this chapter for more about lists; see [Recipe 5.9](#) for more about building and using lists with named elements.

5.7 Selecting List Elements by Position

Problem

You want to access list elements by position.

Solution

Use one of these ways. Here, `lst` is a list variable:

`lst[[n]]`

Selects the n th element from the list

`lst[c(n1, n2, ..., nk)]`

Returns a list of elements, selected by their positions

Note that the first form returns a single element and the second form returns a list.

Discussion

Suppose we have a list of four integers, called `years`:

```
years <- list(1960, 1964, 1976, 1994)
years
#> [[1]]
#> [1] 1960
#>
#> [[2]]
#> [1] 1964
#>
#> [[3]]
#> [1] 1976
#>
#> [[4]]
#> [1] 1994
```

We can access single elements using the double-square-bracket syntax:

```
years[[1]]
#> [1] 1960
```

We can extract sublists using the single-square-bracket syntax:

```
years[c(1, 2)]
#> [[1]]
#> [1] 1960
#>
#> [[2]]
#> [1] 1964
```

This syntax can be confusing because of a subtlety: there is an important difference between `lst[[n]]` and `lst[n]`. They are not the same thing:

`lst[[n]]`

This is an element, not a list. It is the n th element of `lst`.

`lst[n]`

This is a list, not an element. The list contains one element, taken from the n th element of `lst`.



The second form is a special case of `lst[c(n1, n2, ..., nk)]` in which we eliminated the `c(...)` construct because there is only one n .

The difference becomes apparent when we inspect the structure of the result—one is a number and the other is a list:

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

class(years[1])
#> [1] "list"
```

The difference becomes annoyingly apparent when we `cat` the value. Recall that `cat` can print atomic values or vectors but complains about printing structured objects:

```
cat(years[[1]], "\n")
#> 1960

cat(years[1], "\n")
#> Error in cat(years[1], "\n"): argument 1 (type 'list')
#> cannot be handled by 'cat'
```

We got lucky here because R alerted us to the problem. In other contexts, you might work long and hard to figure out that you accessed a sublist when you wanted an element, or vice versa.

5.8 Selecting List Elements by Name

Problem

You want to access list elements by their names.

Solution

Use one of these forms. Here, `lst` is a list variable:

```
lst[["name"]]
```

Selects the element called *name*. Returns NULL if no element has that name.

```
lst$name
```

Same as previous, just different syntax.

```
lst[c(name1, name2, ..., namek)]
```

Returns a list built from the indicated elements of *lst*.

Note that the first two forms return an element, whereas the third form returns a list.

Discussion

Each element of a list can have a name. If named, the element can be selected by its name. This assignment creates a list of four named integers:

```
years <- list(Kennedy = 1960, Johnson = 1964,  
               Carter = 1976, Clinton = 1994)
```

These next two expressions return the same value—namely, the element that is named “Kennedy”:

```
years[["Kennedy"]]  
#> [1] 1960  
years$Kennedy  
#> [1] 1960
```

The following two expressions return sublists extracted from *years*:

```
years[c("Kennedy", "Johnson")]  
#> $Kennedy  
#> [1] 1960  
#>  
#> $Johnson  
#> [1] 1964  
  
years["Carter"]  
#> $Carter  
#> [1] 1976
```

Just as with selecting list elements by position (see [Recipe 5.7](#)), there is an important difference between `lst[["name"]]` and `lst["name"]`. They are not the same:

```
lst[["name"]]
```

This is an element, not a list.

```
lst["name"]
```

This is a list, not an element.



The second form is a special case of `lst[c(name1, name2, ..., namek)]` in which we don't need the `c(...)` construct because there is only one name.

See Also

See [Recipe 5.7](#) to access elements by position rather than by name.

5.9 Building a Name/Value Association List

Problem

You want to create a list that associates names and values, like a dictionary, hash, or lookup table would in another programming language.

Solution

The `list` function lets you give names to elements, creating an association between each name and its value:

```
lst <- list(mid = 0.5, right = 0.841, far.right = 0.977)
```

If you have parallel vectors of names and values, you can create an empty list and then populate the list by using a vectorized assignment statement:

```
values <- c(1, 2, 3)
names <- c("a", "b", "c")
lst <- list()
lst[names] <- values
```

Discussion

Each element of a list can be named, and you can retrieve list elements by name. This gives you a basic programming tool: the ability to associate names with values.

You can assign element names when you build the list. The `list` function allows arguments of the form `name=value`:

```
lst <- list(
  far.left = 0.023,
  left = 0.159,
  mid = 0.500,
  right = 0.841,
  far.right = 0.977
)
lst
#> $far.left
#> [1] 0.023
```

```
#>
#> $left
#> [1] 0.159
#>
#> $mid
#> [1] 0.5
#>
#> $right
#> [1] 0.841
#>
#> $far.right
#> [1] 0.977
```

One way to name the elements is to create an empty list and then populate it via assignment statements:

```
lst <- list()
lst$far.left <- 0.023
lst$left <- 0.159
lst$mid <- 0.500
lst$right <- 0.841
lst$far.right <- 0.977
```

Sometimes you have a vector of names and a vector of corresponding values:

```
values <- -2:2
names <- c("far.left", "left", "mid", "right", "far.right")
```

You can associate the names and the values by creating an empty list and then populating it with a vectorized assignment statement:

```
lst <- list()
lst[names] <- values
lst
#> $far.left
#> [1] -2
#>
#> $left
#> [1] -1
#>
#> $mid
#> [1] 0
#>
#> $right
#> [1] 1
#>
#> $far.right
#> [1] 2
```

Once the association is made, the list can “translate” names into values through a simple list lookup:

```
cat("The left limit is", lst[["left"]], "\n")
#> The left limit is -1
```

```
cat("The right limit is", lst[["right"]], "\n")
#> The right limit is 1

for (nm in names(lst)) cat("The", nm, "limit is", lst[[nm]], "\n")
#> The far.left limit is -2
#> The left limit is -1
#> The mid limit is 0
#> The right limit is 1
#> The far.right limit is 2
```

5.10 Removing an Element from a List

Problem

You want to remove an element from a list.

Solution

Assign `NULL` to the element. R will remove it from the list.

Discussion

To remove a list element, select it by position or by name, and then assign `NULL` to the selected element:

```
years <- list(Kennedy = 1960, Johnson = 1964,
               Carter = 1976, Clinton = 1994)
years
#> $Kennedy
#> [1] 1960
#>
#> $Johnson
#> [1] 1964
#>
#> $Carter
#> [1] 1976
#>
#> $Clinton
#> [1] 1994
years[["Johnson"]] <- NULL # Remove the element labeled "Johnson"
years
#> $Kennedy
#> [1] 1960
#>
#> $Carter
#> [1] 1976
#>
#> $Clinton
#> [1] 1994
```

You can remove multiple elements this way, too:

```
years[c("Carter", "Clinton")] <- NULL # Remove two elements
years
#> $Kennedy
#> [1] 1960
```

5.11 Flattening a List into a Vector

Problem

You want to flatten all the elements of a list into a vector.

Solution

Use the `unlist` function.

Discussion

There are many contexts that require a vector. Basic statistical functions work on vectors but not on lists, for example. If `iq.scores` is a list of numbers, then we cannot directly compute their mean:

```
iq.scores <- list(100, 120, 103, 80, 99)
mean(iq.scores)
#> Warning in mean.default(iq.scores): argument is not numeric or logical:
#> returning NA
#> [1] NA
```

Instead, we must flatten the list into a vector using `unlist` and then compute the mean of the result:

```
mean(unlist(iq.scores))
#> [1] 100
```

Here is another example. We can `cat` scalars and vectors, but we cannot `cat` a list:

```
cat(iq.scores, "\n")
#> Error in cat(iq.scores, "\n"): argument 1 (type 'list') cannot be
#> handled by 'cat'
```

One solution is to flatten the list into a vector before printing:

```
cat("IQ Scores:", unlist(iq.scores), "\n")
#> IQ Scores: 100 120 103 80 99
```

See Also

Conversions such as this are discussed more fully in [Recipe 5.29](#).

5.12 Removing NULL Elements from a List

Problem

Your list contains NULL values. You want to remove them.

Solution

The `compact` function from the `purrr` package will remove the NULL elements.

Discussion

The curious reader may be wondering how a list can contain NULL elements, given that we remove elements by setting them to NULL (see [Recipe 5.10](#)). The answer is that we can *create* a list containing NULL elements:

```
library(purrr)      # or library(tidyverse)

lst <- list("Moe", NULL, "Curly")
lst
#> [[1]]
#> [1] "Moe"
#>
#> [[2]]
#> NULL
#>
#> [[3]]
#> [1] "Curly"

compact(lst)    # Remove NULL element
#> [[1]]
#> [1] "Moe"
#>
#> [[2]]
#> [1] "Curly"
```

In practice, we might also end up with NULL items in a list after applying some transformation.

Note that in R, NA and NULL are not the same thing. The `compact` function will remove NULL from a list but not NA. To remove NA values, see [Recipe 5.13](#).

See Also

See [Recipe 5.10](#) for how to remove list elements and [Recipe 5.13](#) for how to remove list elements conditionally.

5.13 Removing List Elements Using a Condition

Problem

You want to remove elements from a list according to a conditional test, such as removing elements that are undefined, negative, or smaller than some threshold.

Solution

Start with a function that returns TRUE when your criteria are met and FALSE otherwise. Then use the `discard` function from `purrr` to remove values that match your criteria. This code snippet, for example, uses the `is.na` function to remove NA values from `lst`:

```
lst <- list(NA, 0, NA, 1, 2)

lst %>%
  discard(is.na)
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 1
#>
#> [[3]]
#> [1] 2
```

Discussion

The `discard` function removes elements from a list using a *predicate*, which is a function that returns either TRUE or FALSE. The predicate is applied to each element of the list. If the predicate returns TRUE, the element is discarded; otherwise, it is kept.

Suppose we want to remove character strings from `lst`. The function `is.character` is a predicate that returns TRUE if its argument is a character string, so we can use it with `discard`:

```
lst <- list(3, "dog", 2, "cat", 1)

lst %>%
  discard(is.character)
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 1
```

You can define your own predicate and use it with `discard`. This example removes both NA and NULL values from a list by defining the predicate `is_na_or_null`:

```
is_na_or_null <- function(x) {
  is.na(x) || is.null(x)
}

lst <- list(1, NA, 2, NULL, 3)

lst %>%
  discard(is_na_or_null)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Lists can hold complex objects, too, not just atomic values. Suppose that `mods` is a list of linear models created by the `lm` function:

```
mods <- list(lm(x ~ y1),
             lm(x ~ y2),
             lm(x ~ y3))
```

We can define a predicate, `filter_r2`, to identify models whose R^2 values are less than 0.70, then use the predicate to remove those models from `mods`:

```
filter_r2 <- function(model) {
  summary(model)$r.squared < 0.7
}

mods %>%
  discard(filter_r2)
```

The inverse of `discard` is the `keep` function, which uses a predicate to *retain* list elements instead of discarding them.

See Also

See [Recipe 5.7](#), [Recipe 5.10](#), and [Recipe 15.3](#).

5.14 Initializing a Matrix

Problem

You want to create a matrix and initialize it from given values.

Solution

Capture the data in a vector or list, and then use the `matrix` function to shape the data into a matrix. This example shapes a vector into a 2×3 matrix (i.e., two rows and three columns):

```
vec <- 1:6
matrix(vec, 2, 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

Discussion

The first argument of `matrix` is the data, the second argument is the number of rows, and the third argument is the number of columns. Note that the matrix in the Solution was filled column by column, not row by row.

It's common to initialize an entire matrix to one value, such as `0` or `NA`. If the first argument of `matrix` is a single value, then R will apply the Recycling Rule and automatically replicate the value to fill the entire matrix:

```
matrix(0, 2, 3) # Create an all-zeros matrix
#>      [,1] [,2] [,3]
#> [1,]    0    0    0
#> [2,]    0    0    0

matrix(NA, 2, 3) # Create a matrix populated with NAs
#>      [,1] [,2] [,3]
#> [1,]   NA   NA   NA
#> [2,]   NA   NA   NA
```

You can create a matrix with a one-liner, of course, but it becomes difficult to read:

```
mat <- matrix(c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3), 2, 3)
mat
#>      [,1] [,2] [,3]
#> [1,]  1.1  1.3  2.2
#> [2,]  1.2  2.1  2.3
```

A common idiom in R is typing the data itself in a rectangular shape that reveals the matrix structure:

```
theData <- c(
  1.1, 1.2, 1.3,
  2.1, 2.2, 2.3
)
mat <- matrix(theData, 2, 3, byrow = TRUE)
mat
#>      [,1] [,2] [,3]
#> [1,]  1.1  1.2  1.3
#> [2,]  2.1  2.2  2.3
```

Setting `byrow=TRUE` tells `matrix` that the data is row-by-row and not column-by-column (which is the default). In condensed form, that becomes:

```
mat <- matrix(c(1.1, 1.2, 1.3,
                2.1, 2.2, 2.3),
                2, 3,
                byrow = TRUE)
```

Expressed this way, it's easy to see the two rows and three columns of data.

There is a quick-and-dirty way to turn a vector into a matrix: just assign dimensions to the vector. This was discussed in the introduction to this chapter. The following example creates a vanilla vector and then shapes it into a 2×3 matrix:

```
v <- c(1.1, 1.2, 1.3, 2.1, 2.2, 2.3)
dim(v) <- c(2, 3)
v
#>      [,1] [,2] [,3]
#> [1,]  1.1  1.3  2.2
#> [2,]  1.2  2.1  2.3
```

We find this more opaque than using `matrix`, especially since there is no `byrow` option here.

See Also

See [Recipe 5.3](#).

5.15 Performing Matrix Operations

Problem

You want to perform matrix operations such as transposition, inversion, multiplication, or constructing an identity matrix.

Solution

Perform these operations with the following functions:

`t(A)`
Matrix transposition of A

`solve(A)`
Matrix inverse of A

`A %*% B`
Matrix multiplication of A and B

```
diag(n)
Constructs an  $n \times n$  diagonal (identity) matrix
```

Discussion

Recall that $A * B$ is element-wise multiplication, whereas $A \%*\% B$ is matrix multiplication (see [Recipe 2.11](#)).

All these functions return a matrix. Their arguments can be either matrices or data frames. If they are data frames, then R will first convert them to matrices (although this is useful only if the data frame contains exclusively numeric values).

5.16 Giving Descriptive Names to the Rows and Columns of a Matrix

Problem

You want to assign descriptive names to the rows or columns of a matrix.

Solution

Every matrix has a `rownames` attribute and a `colnames` attribute. Assign a vector of character strings to the appropriate attribute:

```
rownames(mat) <- c("rowname1", "rowname2", ..., "rownameN")
colnames(mat) <- c("colname1", "colname2", ..., "colnameN")
```

Discussion

R lets you assign names to the rows and columns of a matrix, which is useful for printing the matrix. R will display the names if they are defined, enhancing the readability of your output. Consider this matrix of correlations between the stock prices of IBM, Microsoft, and Google:

```
print(corr_mat)
#>      [,1]  [,2]  [,3]
#> [1,] 1.000 0.556 0.390
#> [2,] 0.556 1.000 0.444
#> [3,] 0.390 0.444 1.000
```

In this form, the interpretation of the matrix is not self-evident. We can give names to the rows and columns, clarifying its meaning:

```
colnames(corr_mat) <- c("AAPL", "MSFT", "GOOG")
rownames(corr_mat) <- c("AAPL", "MSFT", "GOOG")
corr_mat
#>      AAPL  MSFT  GOOG
#> AAPL  1.000 0.556 0.390
```

```
#> MSFT 0.556 1.000 0.444  
#> GOOG 0.390 0.444 1.000
```

Now you can see at a glance which rows and columns apply to which stocks.

Another advantage of naming rows and columns is that you can refer to matrix elements by those names:

```
# What is the correlation between MSFT and GOOG?  
corr_mat["MSFT", "GOOG"]  
#> [1] 0.444
```

5.17 Selecting One Row or Column from a Matrix

Problem

You want to select a single row or a single column from a matrix.

Solution

The solution depends on what you want. If you want the result to be a simple vector, just use normal indexing:

```
mat[1, ]    # First row  
mat[, 3]    # Third column
```

If you want the result to be a one-row matrix or a one-column matrix, then include the `drop=FALSE` argument:

```
mat[1, , drop=FALSE]  # First row, one-row matrix  
mat[, 3, drop=FALSE]  # Third column, one-column matrix
```

Discussion

Normally, when you select one row or column from a matrix, R strips off the dimensions. The result is a dimensionless vector:

```
mat[1, ]  
#> [1] 1.1 1.2 1.3  
mat[, 3]  
#> [1] 1.3 2.3
```

When you include the `drop=FALSE` argument, however, R retains the dimensions. In that case, selecting a row returns a row vector (a $1 \times n$ matrix):

```
mat[1, , drop=FALSE]  
#>      [,1] [,2] [,3]  
#> [1,] 1.1 1.2 1.3
```

Likewise, selecting a column with `drop=FALSE` returns a column vector (an $n \times 1$ matrix):

```
mat[, 3, drop=FALSE]
#>      [,1]
#> [1,] 1.3
#> [2,] 2.3
```

5.18 Initializing a Data Frame from Column Data

Problem

Your data is organized by columns, and you want to assemble it into a data frame.

Solution

If your data is captured in several vectors and/or factors, use the `data.frame` function to assemble them into a data frame:

```
df <- data.frame(v1, v2, v3, f1)
```

If your data is captured in a list that contains vectors and/or factors, use `as.data.frame` instead:

```
df <- as.data.frame(list.of.vectors)
```

Discussion

A data frame is a collection of columns, each of which corresponds to an observed variable (in the statistical sense, not the programming sense). If your data is already organized into columns, then it's easy to build a data frame.

The `data.frame` function can construct a data frame from vectors, where each vector is one observed variable. Suppose you have two numeric variables, one character variable, and one response variable. The `data.frame` function can create a data frame from your vectors:

```
data.frame(pred1, pred2, pred3, resp)
#>   pred1 pred2 pred3 resp
#> 1  1.75 11.8    AM 13.2
#> 2  4.01 10.7    PM 12.9
#> 3  2.64 12.2    AM 13.9
#> 4  6.03 12.2    PM 14.9
#> 5  2.78 15.0    PM 16.4
```

Notice that `data.frame` takes the column names from your program variables. You can override that default by supplying explicit column names:

```
data.frame(p1 = pred1, p2 = pred2, p3 = pred3, r = resp)
#>   p1   p2 p3    r
#> 1 1.75 11.8 AM 13.2
#> 2 4.01 10.7 PM 12.9
#> 3 2.64 12.2 AM 13.9
```

```
#> 4 6.03 12.2 PM 14.9  
#> 5 2.78 15.0 PM 16.4
```

If you'd rather have a tibble than a data frame, use the `tibble` function from the tidyverse:

```
tibble(p1 = pred1, p2 = pred2, p3 = pred3, r = resp)  
#> # A tibble: 5 x 4  
#>   p1     p2    p3      r  
#>   <dbl> <dbl> <fct> <dbl>  
#> 1 1.75  11.8 AM  13.2  
#> 2 4.01   10.7 PM  12.9  
#> 3 2.64   12.2 AM  13.9  
#> 4 6.03   12.2 PM  14.9  
#> 5 2.78   15.0 PM  16.4
```

Sometimes, your data may indeed be organized into vectors, but those vectors are held in a list, not individual program variables:

```
list.of.vectors <- list(p1=pred1, p2=pred2, p3=pred3, r=resp)
```

In that case, use the `as.data.frame` function to create a data frame from the list:

```
as.data.frame(list.of.vectors)  
#>   p1     p2    p3      r  
#> 1 1.75  11.8 AM  13.2  
#> 2 4.01   10.7 PM  12.9  
#> 3 2.64   12.2 AM  13.9  
#> 4 6.03   12.2 PM  14.9  
#> 5 2.78   15.0 PM  16.4
```

or use `as_tibble` to create a tibble:

```
as_tibble(list.of.vectors)  
#> # A tibble: 5 x 4  
#>   p1     p2    p3      r  
#>   <dbl> <dbl> <fct> <dbl>  
#> 1 1.75  11.8 AM  13.2  
#> 2 4.01   10.7 PM  12.9  
#> 3 2.64   12.2 AM  13.9  
#> 4 6.03   12.2 PM  14.9  
#> 5 2.78   15.0 PM  16.4
```

Factors in data frames

There is an important difference between creating a data frame and creating a tibble. When you use the `data.frame` function to create a data frame, R will convert character values into factors by default. The `pred3` value in the preceding `data.frame` example was converted to a factor, but that is not evident from the output.

The `tibble` and `as_tibble` functions, however, do not change character data. If you look at the `tibble` example, you'll see column `p3` has type `chr`, meaning character.

This difference is something you should be aware of. It can be maddeningly frustrating to debug an issue caused by this subtle difference.

5.19 Initializing a Data Frame from Row Data

Problem

Your data is organized by rows, and you want to assemble it into a data frame.

Solution

Store each row in a one-row data frame. Use `rbind` to bind the rows into one large data frame:

```
rbind(row1, row2, ... , rowN)
```

Discussion

Data often arrives as a collection of observations. Each observation is a record or tuple that contains several values, one for each observed variable. The lines of a flat file are usually like that: each line is one record, each record contains several columns, and each column is a different variable (see [Recipe 4.15](#)). Such data is organized by *observation*, not by *variable*. In other words, you are given rows one at a time rather than columns one at a time.

Each such row might be stored in several ways. One obvious way is as a vector. If you have purely numerical data, use a vector.

Many datasets, however, are a mixture of numeric, character, and categorical data, in which case a vector won't work. We recommend storing each such heterogeneous row in a one-row data frame. (You could store each row in a list, but this recipe gets a little more complicated.)

We need to bind together those rows into a data frame. That's what the `rbind` function does. It binds its arguments in such a way that each argument becomes one row in the result. If we `rbind` these three observations, for example, we get a three-row data frame:

```
r1 <- data.frame(a = 1, b = 2, c = "X")
r2 <- data.frame(a = 3, b = 4, c = "Y")
r3 <- data.frame(a = 5, b = 6, c = "Z")
rbind(r1, r2, r3)
#>   a b c
#> 1 1 2 X
#> 2 3 4 Y
#> 3 5 6 Z
```

When you're working with a large number of rows, they will likely be stored in a list; that is, you will have a list of rows. The `bind_rows` function, from the tidyverse package `dplyr`, handles that case, as shown in this toy example:

```
list.of.rows <- list(r1, r2, r3)
bind_rows(list.of.rows)
#> Warning in bind_rows_(x, .id): Unequal factor levels: coercing to character
#> Warning in bind_rows_(x, .id): binding character and factor vector,
#> coercing into character vector

#> Warning in bind_rows_(x, .id): binding character and factor vector,
#> coercing into character vector

#> Warning in bind_rows_(x, .id): binding character and factor vector,
#> coercing into character vector
#> a b c
#> 1 1 2 X
#> 2 3 4 Y
#> 3 5 6 Z
```

Sometimes, for reasons beyond your control, each row of data is stored in a list rather than one-row data frames. You may be dealing with rows returned by a function or a database package, for example. `bind_rows` can handle that situation as well:

```
# Same toy data, but rows stored in lists
l1 <- list(a = 1, b = 2, c = "X")
l2 <- list(a = 3, b = 4, c = "Y")
l3 <- list(a = 5, b = 6, c = "Z")
list.of.lists <- list(l1, l2, l3)

bind_rows(list.of.lists)
#> # A tibble: 3 x 3
#>   a     b     c
#>   <dbl> <dbl> <chr>
#> 1     1     2 X
#> 2     3     4 Y
#> 3     5     6 Z
```

Factors in data frames

If you would rather get characters instead of factors, you have a couple of options. One is to set the `stringsAsFactors` parameter to `FALSE` when `data.frame` is called:

```
data.frame(a = 1, b = 2, c = "a", stringsAsFactors = FALSE)
#> a b c
#> 1 1 2 a
```

Of course, if you inherited your data and it's already in a data frame with factors, you can convert all the factors to characters using this bonus recipe:

```
# same setup as in the previous examples
l1 <- list( a=1, b=2, c='X' )
```

```

l2 <- list( a=3, b=4, c='Y' )
l3 <- list( a=5, b=6, c='Z' )
obs <- list(l1, l2, l3)
df <- do.call(rbind, Map(as.data.frame, obs))

# Yes, you could use stringsAsFactors=FALSE above,
# but we're assuming the data.frame
# came to you with factors already

i <- sapply(df, is.factor)           # determine which columns are factors
df[i] <- lapply(df[i], as.character) # turn only the factors to characters

```

Keep in mind that if you use a tibble instead of a data frame, then characters will not be forced into factors by default.

See Also

See [Recipe 5.18](#) if your data is organized by columns, not rows.

5.20 Appending Rows to a Data Frame

Problem

You want to append one or more new rows to a data frame.

Solution

Create a second, temporary data frame containing the new rows. Then use the `rbind` function to append the temporary data frame to the original data frame.

Discussion

Suppose we have a data frame of Chicago-area suburbs:

```

suburbs <- read_csv("./data/suburbs.txt")
#> Parsed with column specification:
#> cols(
#>   city = col_character(),
#>   county = col_character(),
#>   state = col_character(),
#>   pop = col_double()
#> )

```

Further suppose we want to append a new row. First, we create a one-row data frame with the new data:

```

newRow <- data.frame(city = "West Dundee", county = "Kane",
                      state = "IL", pop = 7352)

```

Next, we use the `rbind` function to append that one-row data frame to our existing data frame:

```
rbind(suburbs, newRow)
#> # A tibble: 18 x 4
#>   city    county   state   pop
#>   <chr>   <chr>    <chr>   <dbl>
#> 1 Chicago Cook     IL     2853114
#> 2 Kenosha Kenosha  WI     90352
#> 3 Aurora Kane     IL     171782
#> 4 Elgin   Kane     IL     94487
#> 5 Gary    Lake(IN)  IN     102746
#> 6 Joliet  Kendall  IL     106221
#> # ... with 12 more rows
```

The `rbind` function tells R that we are appending a new row to `suburbs`, not a new column. It may be obvious to you that `newRow` is a row and not a column, but it is not obvious to R. (Use the `cbind` function to append a column.)



The new row must use the same column names as the data frame.
Otherwise, `rbind` will fail.

We can combine these two steps into one, of course:

```
rbind(suburbs,
      data.frame(city = "West Dundee", county = "Kane",
                 state = "IL", pop = 7352))
#> # A tibble: 18 x 4
#>   city    county   state   pop
#>   <chr>   <chr>    <chr>   <dbl>
#> 1 Chicago Cook     IL     2853114
#> 2 Kenosha Kenosha  WI     90352
#> 3 Aurora Kane     IL     171782
#> 4 Elgin   Kane     IL     94487
#> 5 Gary    Lake(IN)  IN     102746
#> 6 Joliet  Kendall  IL     106221
#> # ... with 12 more rows
```

We can even extend this technique to multiple new rows because `rbind` allows multiple arguments:

```
rbind(suburbs,
      data.frame(city = "West Dundee", county = "Kane",
                 state = "IL", pop = 7352),
      data.frame(city = "East Dundee", county = "Kane",
                 state = "IL", pop = 3192)
)
#> # A tibble: 19 x 4
```

```

#>   city    county    state    pop
#>   <chr>   <chr>     <chr>    <dbl>
#> 1 Chicago Cook      IL       2853114
#> 2 Kenosha Kenosha   WI       90352
#> 3 Aurora  Kane      IL       171782
#> 4 Elgin   Kane      IL       94487
#> 5 Gary    Lake(IN)  IN       102746
#> 6 Joliet  Kendall   IL       106221
#> # ... with 13 more rows

```

It's worth noting that in the previous examples we seamlessly commingled tibbles and data frames. `suburbs` is a tibble because we used the tidy function `read_csv`, which produces tibbles, while `newRow` was created using `data.frame`, which returns a traditional R data frame. And note that the data frames contain factors while the tibbles do not:

```

str(suburbs) # a tibble
#> Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 17 obs. of
#> 4 variables:
#> $ city : chr "Chicago" "Kenosha" "Aurora" "Elgin" ...
#> $ county: chr "Cook" "Kenosha" "Kane" "Kane" ...
#> $ state : chr "IL" "WI" "IL" "IL" ...
#> $ pop   : num 2853114 90352 171782 94487 102746 ...
#> - attr(*, "spec")=
#>   .. cols(
#>     .. city = col_character(),
#>     .. county = col_character(),
#>     .. state = col_character(),
#>     .. pop = col_double()
#>   .. )
str(newRow) # a data.frame
#> 'data.frame': 1 obs. of 4 variables:
#> $ city : Factor w/ 1 level "West Dundee": 1
#> $ county: Factor w/ 1 level "Kane": 1
#> $ state : Factor w/ 1 level "IL": 1
#> $ pop   : num 7352

```

When the inputs to `rbind` are a mix of `data.frame` objects and `tibble` objects, the result will have the same type as the *first* argument of `rbind`. So this would produce a tibble:

```
rbind(some_tibble, some_data.frame)
```

while this would produce a data frame:

```
rbind(some_data.frame, some_tibble)
```

5.21 Selecting Data Frame Columns by Position

Problem

You want to select columns from a data frame according to their position.

Solution

Use the `select` function:

```
df %>% select(n1, n2, ..., nk)
```

where `df` is a data frame and n_1, n_2, \dots, n_k are integers with values between 1 and the number of columns.

Discussion

Let's use the first three rows of the dataset of population data for the 16 largest cities in the Chicago metropolitan area:

```
suburbs <- read_csv("data/suburbs.txt") %>% head(3)
#> Parsed with column specification:
#> cols(
#>   city = col_character(),
#>   county = col_character(),
#>   state = col_character(),
#>   pop = col_double()
#> )
suburbs
#> # A tibble: 3 x 4
#>   city     county   state    pop
#>   <chr>    <chr>    <chr>   <dbl>
#> 1 Chicago   Cook     IL      2853114
#> 2 Kenosha  Kenosha  WI      90352
#> 3 Aurora   Kane     IL      171782
```

Right off the bat, we can see this is a tibble. This will extract the first column (and only the first column):

```
suburbs %>%
  dplyr::select(1)
#> # A tibble: 3 x 1
#>   city
#>   <chr>
#> 1 Chicago
#> 2 Kenosha
#> 3 Aurora
```

These will extract multiple columns:

```
suburbs %>%
  dplyr::select(1, 3, 4)
#> # A tibble: 3 x 3
#>   city     state    pop
#>   <chr>    <chr>    <dbl>
#> 1 Chicago  IL      2853114
#> 2 Kenosha WI      90352
#> 3 Aurora  IL      171782
suburbs %>%
  dplyr::select(2:4)
#> # A tibble: 3 x 3
#>   county   state    pop
#>   <chr>    <chr>    <dbl>
#> 1 Cook     IL      2853114
#> 2 Kenosha WI      90352
#> 3 Kane     IL      171782
```

List expressions

The `select` verb is part of the tidyverse package `dplyr`. Base R also has its own rich functionality for selecting columns, at the cost of some additional syntax. The choices can be confusing until you understand the logic behind the alternatives.

One alternative uses list expressions. This might seem odd until you recall that a data frame is a list of columns. The list expression selects columns from that list. As you read this explanation, notice how the change in syntax—double brackets versus single brackets—changes the meaning of the expression.

We can select exactly one column by using double brackets ([[and]]):

`df[[n]]`

Returns a *vector*—specifically, the vector in the n th column of `df`

We can select one or more columns by using single brackets ([and]).

`df[n]`

Returns a *data frame* consisting solely of the n th column of `df`

`df[c(n1, n2, ..., nk)]`

Returns a *data frame* built from the columns in positions n_1, n_2, \dots, n_k of `df`

For example, we can use list notation to select the first column from `suburbs`, the `city` column:

```
suburbs[[1]]
#> [1] "Chicago" "Kenosha" "Aurora"
```

That column is a character vector, so that's what `suburbs[[1]]` returns: a vector.

The result changes when we use the single-bracket notation, as in `suburbs[1]` or `suburbs[c(1,3)]`. We still get the requested columns, but R leaves them in a data frame. This example returns the first column as a one-column data frame:

```
suburbs[1]
#> # A tibble: 3 x 1
#>   city
#>   <chr>
#> 1 Chicago
#> 2 Kenosha
#> 3 Aurora
```

And this example returns the first and third columns as a data frame:

```
suburbs[c(1, 3)]
#> # A tibble: 3 x 2
#>   city     state
#>   <chr>    <chr>
#> 1 Chicago  IL
#> 2 Kenosha WI
#> 3 Aurora  IL
```



The expression `suburbs[1]` is actually a shortened form of `suburbs[c(1)]`. We don't need the `c(...)` wrapper because there is only one n .

A major source of confusion is that `suburbs[[1]]` and `suburbs[1]` look similar but produce very different results:

```
suburbs[[1]]
Returns one column
```

```
suburbs[1]
Returns a data frame that contains exactly one column
```

The point here is that “one column” is different from “a data frame that contains one column.” The first expression returns a vector. The second expression returns a data frame, which is a different data structure.

Matrix-style subscripting

You can use matrix-style subscripting to select columns from a data frame:

```
df[, n]
Returns a vector taken from the  $n$ th column (assuming that  $n$  contains exactly
one value)
```

```
df[, c(n1, n2, ..., nk)]
```

Returns a *data frame* built from the columns in positions n_1, n_2, \dots, n_k

An odd quirk can bite you here: you might get a column vector or you might get a data frame, depending upon how many subscripts you use and whether you are operating on a tibble or a `data.frame`. Tibbles will always return tibbles when you index. However, a `data.frame` may return a vector if you use one index.

In the simple case of one index on a `data.frame` you get a vector, like this:

```
# suburbs is a tibble so we convert for this example
suburbs_df <- as.data.frame(suburbs)
suburbs_df[, 1]
#> [1] "Chicago" "Kenosha" "Aurora"
```

But using the same matrix-style syntax with multiple indexes returns a data frame:

```
suburbs_df[, c(1, 4)]
#>      city    pop
#> 1 Chicago 2853114
#> 2 Kenosha  90352
#> 3 Aurora   171782
```

This creates a problem. Suppose you see this expression in some old R code:

```
df[, vec]
```

Quick, does that return a column or a data frame? Well, it depends. If `vec` contains one value, then you get a column; otherwise, you get a data frame. You cannot tell from the syntax alone.

To avoid this problem, you can include `drop=FALSE` in the subscripts, forcing R to return a data frame:

```
df[, vec, drop = FALSE]
```

Now there is no ambiguity about the returned data structure. It's a data frame.

When all is said and done, using matrix notation to select columns from data frames can be tricky. Use `select` when you can.

See Also

See [Recipe 5.17](#) for more about using `drop=FALSE`.

5.22 Selecting Data Frame Columns by Name

Problem

You want to select columns from a data frame according to their name.

Solution

Use `select` and give it the column names.

```
df %>% select(name1, name2, ..., namek)
```

Discussion

All columns in a data frame must have names. If you know the name, it's usually more convenient and readable to select by name, not by position. Note that you don't put the column names in quotes when using `select`.

The solutions described here are similar to those for [Recipe 5.21](#), where we selected columns by position. The only difference is that here we use column names instead of column numbers. All the observations made in that recipe apply here.

List expressions

The `select` verb is part of the tidyverse. Base R itself also has several rich methods for selecting columns by name, at the cost of some additional syntax.

To select a single column, use one of these list expressions. Note that they use *double* brackets ([[and]]):

```
df[["name"]]
```

Returns one column, the column called *name*

```
df$name
```

Same as previous, just different syntax

To select one or more columns, use these list expressions. Note that they use *single* brackets ([and]):

```
df["name"]
```

Selects one column from a data frame

```
df[c("name1", "name2", ..., "namek")]
```

Selects several columns

Matrix-style subscripting

Base R also allows matrix-style subscripting for selecting one or more columns from a data frame by name:

```
df[, "name"]
```

Returns the named column

```
df[, c("name1", "name2", ..., "namek")]  
Selects several columns in a data frame
```

The matrix-style subscripting can return either a column or a data frame, so be careful how many names you supply. See the comments in [Recipe 5.21](#) for a discussion of this “gotcha” and using `drop=FALSE`.

See Also

See [Recipe 5.21](#) to select by position instead of name.

5.23 Changing the Names of Data Frame Columns

Problem

You want to change the names of a data frame’s columns.

Solution

The `rename` function from the `dplyr` package makes renaming pretty easy:

```
df %>% rename(newname1 = oldname1, ... , newnamen = oldnamen)
```

where `df` is a data frame, `oldnamei` are names of columns in `df`, and `newnamei` are the desired new names.

Note that the argument order is `newname = oldname`.

Discussion

The columns of data frames must have names. You can change them using `rename`:

```
df <- data.frame(V1 = 1:3, V2 = 4:6, V3 = 7:9)  
df %>% rename(tom = V1, dick = V2)  
#>   tom dick V3  
#> 1    1    4  7  
#> 2    2    5  8  
#> 3    3    6  9
```

The column names are stored in an attribute called `colnames`, so another way to rename columns is to change that attribute:

```
colnames(df) <- c("tom", "dick", "V2")  
df  
#>   tom dick V2  
#> 1    1    4  7  
#> 2    2    5  8  
#> 3    3    6  9
```

If you happen to be using `select` to select individual columns, you can rename those columns at the same time:

```
df <- data.frame(V1 = 1:3, V2 = 4:6, V3 = 7:9)
df %>% select(tom = V1, V2)
#>   tom V2
#> 1    1  4
#> 2    2  5
#> 3    3  6
```

The difference between renaming with `select` versus renaming with `rename` is that `rename` will rename what you specify, leaving all other columns intact and unchanged, whereas `select` keeps only the columns you select. In the preceding example, `V3` is dropped because it's not in the `select` statement. Both `select` and `rename` use the same argument order: `newname = oldname`.

See Also

See [Recipe 5.29](#).

5.24 Removing NAs from a Data Frame

Problem

Your data frame contains NA values, which is creating problems for you.

Solution

Use `na.omit` to remove rows that contain any NA values:

```
clean_dfrm <- na.omit(dfrm)
```

Discussion

We frequently stumble upon situations where just a few NA values in a data frame cause everything to fall apart. One solution is simply to remove all rows that contain any NAs. That's what `na.omit` does.

Consider a data frame with embedded NA values:

```
df <- data.frame(
  x = c(1, NA, 3, 4, 5),
  y = c(1, 2, NA, 4, 5)
)
df
#>   x  y
#> 1  1  1
#> 2 NA  2
#> 3  3 NA
```

```
#> 4 4 4  
#> 5 5 5
```

The `cumsum` function should calculate cumulative sums, but it stumbles on the NA values:

```
colSums(df)  
#> x y  
#> NA NA
```

If we remove rows with NA values, `cumsum` can complete its summations:

```
cumsum(na.omit(df))  
#> x y  
#> 1 1 1  
#> 4 5 5  
#> 5 10 10
```

But watch out! The `na.omit` function removes *entire* rows. The non-NA values in those rows also disappear, changing the meaning of “cumulative sum.”

This recipe works for removing NA from vectors and matrices, too, but not lists.

The obvious danger here is that simply dropping observations from your data could render the results numerically or statistically meaningless. Make sure that omitting data makes sense in your context. Remember that `na.omit` will remove entire rows, not just the NA values, which could eliminate useful information.

5.25 Excluding Columns by Name

Problem

You want to exclude a column from a data frame using its name.

Solution

Use the `select` function from the `dplyr` package with a dash (minus sign) in front of the name of the column to exclude:

```
select(df, -bad)  # Select all columns from df except bad
```

Discussion

Placing a minus sign in front of a variable name tells the `select` function to drop that variable.

This can come in handy when we’re calculating a correlation matrix from a data frame, and we want to exclude the nondata columns such as labels:

```
cor(patient_data)
#>           patient_id    pre dosage   post
#> patient_id     1.0000  0.159 -0.0486  0.391
#> pre            0.1590  1.000  0.8104 -0.289
#> dosage          -0.0486  0.810  1.0000 -0.526
#> post            0.3912 -0.289 -0.5262  1.000
```

This correlation matrix includes the meaningless “correlation” between `patient_id` and other variables, which is annoying. We can exclude the `patient_id` column to clean up the output:

```
patient_data %>%
  select(-patient_id) %>%
  cor
#>           pre dosage   post
#> pre        1.000  0.810 -0.289
#> dosage      0.810  1.000 -0.526
#> post       -0.289 -0.526  1.000
```

We can exclude multiple columns the same way:

```
patient_data %>%
  select(-patient_id, -dosage) %>%
  cor()
#>           pre   post
#> pre        1.000 -0.289
#> post      -0.289  1.000
```

5.26 Combining Two Data Frames

Problem

You want to combine the contents of two data frames into one data frame.

Solution

To combine the columns of two data frames side by side, use `cbind` (column bind):

```
all.cols <- cbind(df1, df2)
```

To “stack” the rows of two data frames, use `rbind` (row bind):

```
all.rows <- rbind(df1, df2)
```

Discussion

You can combine data frames in one of two ways: either by putting the columns side by side to create a wider data frame, or by “stacking” the rows to create a taller data frame.

The `cbind` function will combine data frames side by side:

```
df1 <- data.frame(a = c(1,2))
df2 <- data.frame(b = c(7,8))

cbind(df1, df2)
#>   a b
#> 1 1 7
#> 2 2 8
```

You would normally combine columns with the same height (number of rows). Technically speaking, however, `cbind` does not require matching heights. If one data frame is short, R will invoke the Recycling Rule to extend the short columns as necessary (see [Recipe 5.3](#)), which may or may not be what you want.

The `rbind` function will “stack” the rows of two data frames:

```
df1 <- data.frame(x = c("a", "a"), y = c(5, 6))
df2 <- data.frame(x = c("b", "b"), y = c(9, 10))
rbind(df1, df2)
#>   x     y
#> 1 a     5
#> 2 a     6
#> 3 b     9
#> 4 b    10
```

The `rbind` function requires that the data frames have the same width—the same number of columns and same column names. The columns need not be in the same *order*, however; `rbind` will sort that out.

Finally, this recipe is slightly more general than the title implies. First, you can combine more than two data frames because both `rbind` and `cbind` accept multiple arguments. Second, you can apply this recipe to other data types because `rbind` and `cbind` work also with vectors, lists, and matrices.

5.27 Merging Data Frames by Common Column

Problem

You have two data frames that share a common column. You want to merge or join their rows into one data frame by matching on the common column.

Solution

We can use the `join` functions from the `dplyr` package to join our data frames together on a common column. If you want only rows that appear in *both* data frames, use `inner_join`:

```
inner_join(df1, df2, by = "col")
```

where “`col`” is the column that appears in both data frames.

If you want all rows that appear in *either* data frame, use `full_join` instead:

```
full_join(df1, df2, by = "col")
```

If you want all rows from `df1` and only those from `df2` that match, use `left_join`:

```
left_join(df1, df2, by = "col")
```

Or to get all records from `df2` and only the matching ones from `df1`, use `right_join`:

```
right_join(df1, df2, by = "col")
```

Discussion

Suppose we have two data frames, `born` and `died`, that each contain a column called `name`:

```
born <- tibble(  
  name = c("Moe", "Larry", "Curly", "Harry"),  
  year.born = c(1887, 1902, 1903, 1964),  
  place.born = c("Bensonhurst", "Philadelphia", "Brooklyn", "Moscow")  
)  
  
died <- tibble(  
  name = c("Curly", "Moe", "Larry"),  
  year.died = c(1952, 1975, 1975)  
)
```

We can merge them into one data frame by using `name` to combine matched rows:

```
inner_join(born, died, by="name")  
#> # A tibble: 3 x 4  
#>   name  year.born place.born  year.died  
#>   <chr>    <dbl> <chr>        <dbl>  
#> 1 Moe      1887 Bensonhurst     1975  
#> 2 Larry    1902 Philadelphia     1975  
#> 3 Curly    1903 Brooklyn       1952
```

Notice that `inner_join` does not require the rows to be sorted or even to occur in the same order. It found the matching rows for `Curly` even though they occur in different positions. It also discarded the row for `Harry`, which appeared only in `born`.

A `full_join` of these data frames includes every row of both, even rows with no matching values:

```
full_join(born, died, by="name")  
#> # A tibble: 4 x 4  
#>   name  year.born place.born  year.died  
#>   <chr>    <dbl> <chr>        <dbl>  
#> 1 Moe      1887 Bensonhurst     1975  
#> 2 Larry    1902 Philadelphia     1975  
#> 3 Curly    1903 Brooklyn       1952  
#> 4 Harry    1964 Moscow          NA
```

Where a data frame has no matching value, its columns are filled with NA: the year.died for Harry is NA.

If we don't supply the join function with a field to join by, then it will attempt to join by any field with matching names in both data frames and will return an informational response stating which field it is joining on:

```
full_join(born, died)
#> Joining, by = "name"
#> # A tibble: 4 x 4
#>   name  year.born place.born    year.died
#>   <chr>     <dbl> <chr>          <dbl>
#> 1 Moe        1887 Bensonhurst      1975
#> 2 Larry      1902 Philadelphia    1975
#> 3 Curly      1903 Brooklyn       1952
#> 4 Harry      1964 Moscow          NA
```

If we want to join two data frames on a field that does not have the same name in both data frames, we need our by parameter to be a vector of equalities:

```
df1 <- data.frame(key1 = 1:3, value=2)
df2 <- data.frame(key2 = 1:3, value=3)

inner_join(df1, df2, by = c("key1" = "key2"))
#> key1 value.x value.y
#> 1     1       2       3
#> 2     2       2       3
#> 3     3       2       3
```

Notice in the preceding example how both tables have a field named value that gets renamed in the output. The field from the first table becomes value.x, while the field from the second table becomes value.y. `dplyr` joins will always rename output this way when there is a naming clash on columns not being joined on.

See Also

See [Recipe 5.26](#) for other ways to combine data frames.

The example joined on a single column, name, but these functions can join on multiple columns, too. For details, see the function documentation by typing `?dplyr::join`.

These join operations were inspired by SQL. Just like in SQL, there are multiple types of joins in `dplyr`, including inner, left, right, full, semi, and anti. Again, see the function documentation.

5.28 Converting One Atomic Value into Another

Problem

You have a data value that has an atomic data type: character, complex, double, integer, or logical. You want to convert this value into one of the other atomic data types.

Solution

For each atomic data type, there is a function for converting values to that type. The conversion functions for atomic types include:

- `as.character(x)`
- `as.complex(x)`
- `as.numeric(x)` or `as.double(x)`
- `as.integer(x)`
- `as.logical(x)`

Discussion

Converting one atomic type into another is usually pretty simple. If the conversion works, you get what you would expect. If it does not work, you get NA:

```
as.numeric(" 3.14 ")
#> [1] 3.14
as.integer(3.14)
#> [1] 3
as.numeric("foo")
#> Warning: NAs introduced by coercion
#> [1] NA
as.character(101)
#> [1] "101"
```

If you have a vector of atomic types, these functions apply themselves to every value. So the preceding examples of converting scalars generalize easily to converting entire vectors:

```
as.numeric(c("1", "2.718", "7.389", "20.086"))
#> [1] 1.00 2.72 7.39 20.09
as.numeric(c("1", "2.718", "7.389", "20.086", "etc."))
#> Warning: NAs introduced by coercion
#> [1] 1.00 2.72 7.39 20.09    NA
as.character(101:105)
#> [1] "101" "102" "103" "104" "105"
```

When converting logical values into numeric values, R converts FALSE to 0 and TRUE to 1:

```
as.numeric(FALSE)
#> [1] 0
as.numeric(TRUE)
#> [1] 1
```

This behavior is useful when you are counting occurrences of TRUE in vectors of logical values. If `logvec` is a vector of logical values, then `sum(logvec)` does an implicit conversion from logical to integer values and returns the number of TRUES:

```
logvec <- c(TRUE, FALSE, TRUE, TRUE, TRUE, FALSE)
sum(logvec) ## num true
#> [1] 4
length(logvec) - sum(logvec) ## num not true
#> [1] 2
```

5.29 Converting One Structured Data Type into Another

Problem

You want to convert a variable from one structured data type to another—for example, converting a vector into a list, or a matrix into a data frame.

Solution

These functions convert their argument into the corresponding structured data type:

- `as.data.frame(x)`
- `as.list(x)`
- `as.matrix(x)`
- `as.vector(x)`

Some of these conversions may surprise you, however. We suggest you review [Table 5-2](#) for more detail.

Discussion

Converting between structured data types can be tricky. Some conversions behave as you'd expect. If you convert a matrix into a data frame, for instance, the rows and columns of the matrix become the rows and columns of the data frame. No sweat.

In other cases, the results might surprise you. [Table 5-2](#) summarizes some noteworthy examples.

Table 5-2. Data conversions

Conversion	How	Notes
Vector→List	<code>as.list(vec)</code>	Don't use <code>list(vec)</code> ; that creates a one-element list whose only element is a copy of <code>vec</code> .
Vector→Matrix	To create a one-column matrix: <code>cbind(vec)</code> or <code>as.matrix(vec)</code> To create a one-row matrix: <code>rbind(vec)</code> To create an $n \times m$ matrix: <code>matrix(vec, n, m)</code>	See Recipe 5.14 .
Vector→Data frame	To create a one-column data frame: <code>as.data.frame(vec)</code> To create a one-row data frame: <code>as.data.frame(rbind(vec))</code>	
List→Vector	<code>unlist(lst)</code>	Use <code>unlist</code> rather than <code>as.vector</code> ; see Note 1 and Recipe 5.11 .
List→Matrix	To create a one-column matrix: <code>as.matrix(lst)</code> To create a one-row matrix: <code>as.matrix(rbind(lst))</code> To create an $n \times m$ matrix: <code>matrix(lst, n, m)</code>	
List→Data frame	If the list elements are columns of data: <code>as.data.frame(lst)</code> If the list elements are rows of data, see Recipe 5.19 .	
Matrix→Vector	<code>as.vector(mat)</code>	Returns all matrix elements in a vector.
Matrix→List	<code>as.list(mat)</code>	Returns all matrix elements in a list.
Matrix→Data frame	<code>as.data.frame(mat)</code>	
Data frame→Vector	To convert a one-row data frame: <code>df[1,]</code> To convert a one-column data frame: <code>df[, 1]</code> or <code>df[[1]]</code>	See Note 2.
Data frame→List	<code>as.list(df)</code>	See Note 3.
Data frame→Matrix	<code>as.matrix(df)</code>	See Note 4.

The notes cited in the table are as follows:

- When you convert a list into a vector, the conversion works cleanly if your list contains atomic values that are all of the same mode. Things become complicated if either your list contains mixed modes (e.g., numeric and character), in which case everything is converted to characters, or your list contains other structured data types (such as sublists or data frames), in which case very odd things happen, so don't do that.
- Converting a data frame into a vector makes sense only if the data frame contains one row or one column. To extract all its elements into one long vector, use

`as.vector(as.matrix(df))`. But even that makes sense only if the data frame is all numeric or all character; if not, everything is first converted to character strings.

3. Converting a data frame into a list may seem odd in that a data frame is already a list (i.e., a list of columns). Using `as.list` essentially removes the class (`data.frame`) and thereby exposes the underlying list. That is useful when you want R to treat your data structure as a list—say, for printing.
4. Be careful when converting a data frame into a matrix. If the data frame contains only numeric values, then you get a numeric matrix. If it contains only character values, you get a character matrix. But if the data frame is a mix of numbers, characters, and/or factors, then all values are first converted to characters. The result is a matrix of character strings.

Special considerations for matrices

The matrix conversions detailed here assume that your matrix is homogeneous—that is, all elements have the same mode (e.g., all numeric or all character). A matrix can be heterogeneous, too, when the matrix is built from a list. If so, conversions become messy. For example, when you convert a mixed-mode matrix to a data frame, the data frame's columns are actually lists (to accommodate the mixed data).

See Also

See [Recipe 5.28](#) for converting atomic data types; see the introduction to this chapter for remarks on problematic conversions.

Data Transformations

While traditional programming languages use loops, R has traditionally encouraged using vectorized operations and the `apply` family of functions to crunch data in batches, greatly streamlining the calculations. There is nothing to prevent you from writing loops in R that break your data into whatever chunks you want and then doing an operation on each chunk. However, using vectorized functions can, in many cases, increase the speed, readability, and maintainability of your code.

In recent history, though, the tidyverse—specifically the `purrr` and `dplyr` packages—has introduced new idioms into R that make these concepts easier to learn and slightly more consistent. The name `purrr` comes from a play on the phrase “Pure R.” A “pure function” is a function whose result is determined only by its inputs, and which does not produce any side effects. This is not a functional programming concept you need to understand in order to get great value from `purrr`, however. All most users need to know is that `purrr` contains functions to help us operate “chunk by chunk” on our data in a way that meshes well with other tidyverse packages such as `dplyr`.

Base R has many `apply` functions—`apply`, `lapply`, `sapply`, `tapply`, and `mapply`—as well as their cousins, `by` and `split`. These are solid functions that have been workhorses in Base R for years. We struggled a bit with how much to focus on the Base R `apply` functions and how much to focus on the newer “tidy” approach. After much debate, we’ve chosen to try to illustrate the `purrr` approach and to acknowledge Base R approaches and, in a few places, to illustrate both. The interface to `purrr` and `dplyr` is very clean and, we believe, in most cases, more intuitive.

6.1 Applying a Function to Each List Element

Problem

You have a list, and you want to apply a function to each element of the list.

Solution

Use `map` to apply a function to every element of a list:

```
library(tidyverse)  
  
lst %>%  
  map(fun)
```

Discussion

Let's look at a specific example of taking the average of all the numbers in each element of a list:

```
library(tidyverse)  
  
lst <- list(  
  a = c(1,2,3),  
  b = c(4,5,6)  
)  
lst %>%  
  map(mean)  
#> $a  
#> [1] 2  
#>  
#> $b  
#> [1] 5
```

The `map` function will call your function once for every element in your list. Your function should expect one argument, an element from the list. The `map` functions will collect the returned values and return them in a list.

The `purrr` package contains a whole family of `map` functions that take a list or a vector and then return an object with the same number of elements as the input. The type of object they return varies based on which `map` function is used. See the help file for `map` for a complete list, but a few of the most common are as follows:

`map`

Always returns a list, and the elements of the list may be of different types. This is quite similar to the Base R function `lapply`.

`map_chr`

Returns a character vector.

```
map_int
```

Returns an integer vector.

```
map_dbl
```

Returns a floating-point numeric vector.

Let's take a quick look at a contrived situation where we have a function that could result in a character or an integer result:

```
fun <- function(x) {  
  if (x > 1) {  
    1  
  } else {  
    "Less Than 1"  
  }  
}  
  
fun(5)  
#> [1] 1  
fun(0.5)  
#> [1] "Less Than 1"
```

Let's create a list of elements that we can map fun to and look at how some of the `map` variants behave:

```
lst <- list(.5, 1.5, .9, 2)  
  
map(lst, fun)  
#> [[1]]  
#> [1] "Less Than 1"  
#>  
#> [[2]]  
#> [1] 1  
#>  
#> [[3]]  
#> [1] "Less Than 1"  
#>  
#> [[4]]  
#> [1] 1
```

You can see that `map` produced a list and it is of mixed data types.

`map_chr` will produce a character vector and coerce the numbers into characters:

```
map_chr(lst, fun)  
#> [1] "Less Than 1" "1.000000" "Less Than 1" "1.000000"  
  
## or using pipes  
lst %>%  
  map_chr(fun)  
#> [1] "Less Than 1" "1.000000" "Less Than 1" "1.000000"
```

while `map_dbl` will try to coerce a character string into a double and die trying:

```
map_dbl(lst, fun)
#> Error: Can't coerce element 1 from a character to a double
```

As mentioned earlier, the Base R `lapply` function acts very much like `map`. The Base R `sapply` function is more like the other `map` functions we discussed previously, in that the function tries to simplify the results into a vector or matrix.

See Also

See [Recipe 15.3](#).

6.2 Applying a Function to Every Row of a Data Frame

Problem

You have a function and you want to apply it to every row in a data frame.

Solution

The `mutate` function will create a new variable based on a vector of values. But if we are using a function that can't take in a vector and output a vector, then we have to do a row-by-row operation using `rowwise`.

We can use `rowwise` in a pipe chain to tell `dplyr` to do all following commands row by row:

```
df %>%
  rowwise() %>%
  row_by_row_function()
```

Discussion

Let's create a function and apply it row by row to a data frame. Our function will simply calculate the sum of a sequence from `a` to `b` by `c`:

```
fun <- function(a, b, c) {
  sum(seq(a, b, c))
}
```

Let's create some data to apply this function to, then use `rowwise` to apply our function, `fun`, to it:

```
df <- data.frame(mn = c(1, 2, 3),
                  mx = c(8, 13, 18),
                  rng = c(1, 2, 3))

df %>%
  rowwise %>%
  mutate(output = fun(a = mn, b = mx, c = rng))
```

```
#> Source: local data frame [3 x 4]
#> Groups: <by row>
#>
#> # A tibble: 3 x 4
#>   mn     mx    rng output
#>   <dbl> <dbl> <dbl> <dbl>
#> 1     1     8     1    36
#> 2     2    13     2    42
#> 3     3    18     3    63
```

If we tried to run this function without `rowwise`, it would have thrown an error because the `seq` function cannot process an entire vector:

```
df %>%
  mutate(output = fun(a = mn, b = mx, c = rng))
#> Error in seq.default(a, b, c): 'from' must be of length 1
```

6.3 Applying a Function to Every Row of a Matrix

Problem

You have a matrix. You want to apply a function to every row, calculating the function result for each row.

Solution

Use the `apply` function. Set the second argument to `1` to indicate row-by-row application of the function:

```
results <- apply(mat, 1, fun)    # mat is a matrix, fun is a function
```

The `apply` function will call `fun` once for each row of the matrix, assemble the returned values into a vector, and then return that vector.

Discussion

You may notice that we show only the use of the Base R `apply` function here, while other recipes illustrate `purrr` alternatives. As of this writing, matrix operations are out of scope for `purrr`, so we use the very solid Base R `apply` function. If you really like the `purrr` syntax, you can use those functions if you first convert your matrix to a data frame or tibble. But if your matrix is large, you will notice a meaningful runtime slowdown using `purrr`.

Suppose we have a matrix `long` containing longitudinal data, so each row has data for one subject and the columns contain the repeated observations over time:

```
long <- matrix(1:15, 3, 5)
long
#>      [,1] [,2] [,3] [,4] [,5]
```

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

We could calculate the average observation for each subject by applying the `mean` function to each row. The result is a vector:

```
apply(long, 1, mean)  
#> [1] 7 8 9
```

If our matrix has row names, `apply` uses them to identify the elements of the resulting vector, which is handy:

```
rownames(long) <- c("Moe", "Larry", "Curly")  
apply(long, 1, mean)  
#> Moe Larry Curly  
#> 7 8 9
```

The function being called should expect one argument, a vector, which will be one row from the matrix. The function can return a scalar or a vector. In the vector case, `apply` assembles the results into a matrix. The `range` function returns a vector of two elements, the minimum and the maximum, so applying it to `long` produces a matrix:

```
apply(long, 1, range)  
#> Moe Larry Curly  
#> [1,] 1 2 3  
#> [2,] 13 14 15
```

You can employ this recipe on data frames as well. It works if the data frame is homogeneous—that is, either all numbers or all character strings. When the data frame has columns of different types, extracting vectors from the rows isn’t sensible because vectors must be homogeneous.

6.4 Applying a Function to Every Column

Problem

You have a matrix or data frame, and you want to apply a function to every column.

Solution

For a matrix, use the `apply` function. Set the second argument to 2, which indicates column-by-column application of the function. So, if our matrix or data frame was named `mat` and we wanted to apply a function named `fun` to every column, it would look like this:

```
apply(mat, 2, fun)
```

For a data frame, use the `map_df` function from `purrr`:

```
df2 <- map_df(df, fun)
```

Discussion

Let's look at an example with real numbers and apply the `mean` function to every column of a matrix:

```
mat <- matrix(c(1, 3, 2, 5, 4, 6), 2, 3)
colnames(mat) <- c("t1", "t2", "t3")
mat
#>      t1 t2 t3
#> [1,]  1  2  4
#> [2,]  3  5  6

apply(mat, 2, mean) # Compute the mean of every column
#> t1  t2  t3
#> 2.0 3.5 5.0
```

In Base R, the `apply` function is intended for processing a matrix or data frame. The second argument of `apply` determines the direction:

- 1 means process row by row.
- 2 means process column by column.

This is more mnemonic than it looks. We speak of matrices in “rows and columns,” so rows are first and columns second: 1 and 2, respectively.

A data frame is a more complicated data structure than a matrix, so there are more options. You can simply use `apply`, in which case R will convert your data frame to a matrix and then apply your function. That will work if your data frame contains only one type of data but will probably not do what you want if some columns are numeric and some are character. In that case, R will force all columns to have identical types, likely performing an unwanted conversion as a result.

Fortunately, there are multiple alternatives. Recall that a data frame is a kind of list: it is a list of the columns of the data frame. `purrr` has a whole family of `map` functions that return different types of objects. Of particular interest here is `map_df`, which returns a `data.frame` (thus the `df` in the name):

```
df2 <- map_df(df, fun) # Returns a data.frame
```

The function `fun` should expect one argument: a column from the data frame.

Here is a common recipe to check the types of columns in data frames. In this example, the `batch` column of this data frame, at a quick glance, seems to contain numbers:

```
load("./data/batches.rdata")
head(batches)
#>   batch clinic dosage shrinkage
#> 1     3    KY    IL   -0.307
#> 2     3    IL    IL  -1.781
#> 3     1    KY    IL   -0.172
#> 4     3    KY    IL   1.215
#> 5     2    IL    IL   1.895
#> 6     2    NJ    IL   -0.430
```

But using `map_df` to print out the class of each column reveals the column `batch` to be a factor instead:

```
map_df(batches, class)
#> # A tibble: 1 x 4
#>   batch clinic dosage shrinkage
#>   <chr>  <chr>  <chr>  <chr>
#> 1 factor factor factor numeric
```



Notice how the third line of the output says `<chr>` repeatedly. This is because the output of `class` is being put in a data frame and then printed. The intermediate data frame is all character fields. It's the last row that tells us our original data frame has three factor columns and one numeric field.

See Also

See [Recipe 5.21](#), [Recipe 6.1](#), and [Recipe 6.3](#).

6.5 Applying a Function to Parallel Vectors or Lists

Problem

You have a function that takes multiple arguments. You want to apply the function element-wise to vectors and obtain a vector result. Unfortunately, the function is not vectorized; that is, it works on scalars but not on vectors.

Solution

Use one of the `map` or `pmap` functions from the tidyverse core package `purrr`. The most general solution is to put your vectors in a list, then use `pmap`:

```
lst <- list(v1, v2, v3)
pmap(lst, fun)
```

`pmap` will take the elements of `lst` and pass them as the inputs to `fun`.

If you only have two vectors you are passing as inputs to your function, the `map2` family of functions is convenient and saves you the step of putting your vectors in a list first. `map2` will return a list:

```
map2(v1, v2, fun)
```

while the typed variants (`map2_chr`, `map2_dbl`, etc.) return vectors of the type their name implies. So, if `fun` returns only a double, use the typed variant of `map2` instead:

```
map2_dbl(v1, v2, fun)
```

The typed variants in `purrr` functions refer to the *output* type expected from the function. All the typed variants return vectors of their respective type, while the untyped variants return lists, which allow mixing of types.

Discussion

The basic operators of R, such as `x + y`, are vectorized; this means that they compute their result element by element and return a vector of results. Also, many R functions are vectorized.

Not all functions are vectorized, however, and those that are not typed work only on scalars. Using vector arguments produces errors at best and meaningless results at worst. In such cases, the `map` functions from `purrr` can effectively vectorize the function for you.

Consider the `gcd` function from [Recipe 15.3](#), which takes two arguments:

```
gcd <- function(a, b) {  
  if (b == 0) {  
    return(a)  
  } else {  
    return(gcd(b, a %% b))  
  }  
}
```

If we apply `gcd` to two vectors, the result is wrong answers and a pile of error messages:

```
gcd(c(1, 2, 3), c(9, 6, 3))  
#> Warning in if (b == 0) {: the condition has length > 1 and only the first  
#> element will be used  
  
#> Warning in if (b == 0) {: the condition has length > 1 and only the first  
#> element will be used  
  
#> Warning in if (b == 0) {: the condition has length > 1 and only the first  
#> element will be used  
#> [1] 1 2 0
```

The function is not vectorized, but we can use `map` to “vectorize” it. In this case, since we have two inputs we’re mapping over, we should use the `map2` function. This gives the element-wise greatest common divisors (GCDs) between two vectors:

```
a <- c(1, 2, 3)
b <- c(9, 6, 3)
my_gcds <- map2(a, b, gcd)
my_gcds
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Notice that `map2` returns a list of lists. If we wanted the output in a vector, we could use `unlist` on the result:

```
unlist(my_gcds)
#> [1] 1 2 3
```

or use one of the typed variants, such as `map2_dbl`.

The `map` family of `purrr` functions give you a series of variations that return specific types of output. The suffixes on the function names communicate the type of vector they will return. While `map` and `map2` return lists, since the type-specific variants are returning objects guaranteed to be the same type, they can be put in atomic vectors. For example, we could use the `map_chr` function to ask R to coerce the results into character output or `map2_dbl` to ensure the results are doubles:

```
map2_chr(a, b, gcd)
#> [1] "1.000000" "2.000000" "3.000000"
map2_dbl(a, b, gcd)
#> [1] 1 2 3
```

If our data has more than two vectors, or the data is already in a list, we can use the `pmap` family of functions, which take a list as an input:

```
lst <- list(a,b)
pmap(lst, gcd)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Or if we want a typed vector as output:

```
lst <- list(a,b)
pmap_dbl(lst, gcd)
#> [1] 1 2 3
```

With the `purrr` functions, remember that the `pmap` family are parallel mappers that take in a *list* as inputs, while `map2` functions take two, and only two, *vectors* as inputs.

See Also

This is really just a special case of our very first recipe in this chapter, [Recipe 6.1](#). See that recipe for more discussion of `map` variants. In addition, Jenny Bryan has a great collection of `purrr` tutorials on her [GitHub site](#).

6.6 Applying a Function to Groups of Data

Problem

Your data elements occur in groups. You want to process the data by groups—for example, summing by group or averaging by group.

Solution

The easiest way to do grouping is with the `dplyr` function `group_by` in conjunction with `summarize`. If our data frame is `df` and has a variable we want to group by named `grouping_var`, and we want to apply the function `fun` to all the combinations of `v1` and `v2`, we can do that with `group_by`:

```
df %>%
  group_by(v1, v2) %>%
  summarize(
    result_var = fun(value_var)
  )
```

Discussion

Let's look at a specific example where our input data frame, `df`, contains a variable, `my_group`, which we want to group by, and a field named `values` which we would like to calculate some statistics on:

```
df <- tibble(
  my_group = c("A", "B", "A", "B", "A", "B"),
  values = 1:6
)

df %>%
  group_by(my_group) %>%
  summarize(
    avg_values = mean(values),
```

```

    tot_values = sum(values),
    count_values = n()
)
#> # A tibble: 2 x 4
#>   my_group avg_values tot_values count_values
#>   <chr>        <dbl>     <int>        <int>
#> 1 A             3         9            3
#> 2 B             4        12            3

```

The output has one record per grouping along with calculated values for the three summary fields we defined.



If you are grouping by several variables, please be aware that `summarize` will change your grouping. Each grouping becomes a single row; at the same time, it also removes the *last* grouping variable. In other words, if you group your data by A, B, and C and then `summarize` it, the resulting data frame is grouped only by A and B. This is surprising but necessary. If `summarize` kept the C grouping, each “group” would contain exactly one row, which would be pointless.

6.7 Creating a New Column Based on Some Condition

Problem

You want to create a new column in a data frame based on some condition.

Solution

Using the `dplyr` tidyverse package, we can create new data frame columns with `mutate` and then use `case_when` to implement conditional logic.

```

df %>%
  mutate(
    new_field = case_when(my_field == "something" ~ "result",
                          my_field != "something else" ~ "other result",
                          TRUE ~ "all other results")
  )

```

Discussion

The `case_when` function from `dplyr` is analogous to CASE WHEN in SQL or nested IF statements in Excel. The function tests every element and, when it finds a condition that is true, returns the value on the righthand side of the ~ (tilde).

Let's look at an example where we want to add a text field that describes a value. First let's set up some simple example data in a data frame with one column named `vals`:

```
df <- data.frame(vals = 1:5)
```

Now let's implement logic that creates a field called `new_vals`. If `vals` is less than or equal to 2, we'll return 2 or less; if the value is greater than 2 and less than or equal to 4, we'll return 2 to 4, and otherwise we'll return over 4:

```
df %>%
  mutate(new_vals = case_when(vals <= 2 ~ "2 or less",
                             vals > 2 & vals <= 4 ~ "2 to 4",
                             TRUE ~ "over 4"))

#>   vals  new_vals
#> 1    1  2 or less
#> 2    2  2 or less
#> 3    3  2 to 4
#> 4    4  2 to 4
#> 5    5  over 4
```

You can see in the example that the condition goes on the left of the `~`, while the resulting return value goes on the right. Each condition is separated by commas. `case_when` will evaluate each condition sequentially and stop evaluating as soon as one of the criteria returns `TRUE`. Our last line is our “or else” statement. Setting the condition to `TRUE` ensures that, no matter what, this condition will be met if no condition above it has returned `TRUE`.

See Also

See [Recipe 6.2](#) for more examples of using `mutate`.

Strings and Dates

Strings? Dates? In a statistical programming package?

As soon as you read files or print reports, you need strings. When you work with real-world problems, you need dates.

R has facilities for both strings and dates. They are clumsy compared to string-oriented languages such as Perl, but then it's a matter of the right tool for the job. We wouldn't want to perform logistic regression in Perl.

Some of this clunkiness with strings and dates has been improved through the tidyverse packages `stringr` and `lubridate`. As with other chapters in this book, the examples here will pull from Base R as well as add-on packages that make life easier, faster, and more convenient.

Classes for Dates and Times

R has a variety of classes for working with dates and times, which is nice if you prefer having a choice but annoying if you prefer living simply. There is a critical distinction among the classes: some are date-only classes, some are datetime classes. All classes can handle calendar dates (e.g., March 15, 2019), but not all can represent a datetime (11:45 AM on March 1, 2019).

The following classes are included in the base distribution of R:

Date

The `Date` class can represent a calendar date but not a clock time. It is a solid, general-purpose class for working with dates, including conversions, formatting, basic date arithmetic, and time-zone handling. Most of the date-related recipes in this book are built on the `Date` class.

`POSIXct`

This is a datetime class, and it can represent a moment in time with an accuracy of one second. Internally, the datetime is stored as the number of seconds since January 1, 1970, so it's a very compact representation. This class is recommended for storing datetime information (e.g., in data frames).

`POSIXlt`

This is also a datetime class, but the representation is stored in a nine-element list that includes the year, month, day, hour, minute, and second. This representation makes it easy to extract date parts, such as the month or hour. Obviously, this is much less compact than the `POSIXct` class; hence, it is normally used for intermediate processing and not for storing data.

The base distribution also provides functions for easily converting between representations: `as.Date`, `as.POSIXct`, and `as.POSIXlt`.

The following helpful packages are available for downloading from CRAN:

`chron`

The `chron` package can represent both dates and times, but without the added complexities of handling time zones and Daylight Saving Time. It's therefore easier to use than `Date` but less powerful than `POSIXct` and `POSIXlt`. It would be useful for work in econometrics or time series analysis.

`lubridate`

This is a tidyverse package designed to make working with dates and times easier while keeping the important bells and whistles such as time zones. It's especially clever regarding datetime arithmetic. This package introduces some helpful constructs like durations, periods, and intervals. `lubridate` is part of the tidyverse, so it is installed when you `install.packages('tidyverse')`, but it is not part of "core tidyverse," so it does not get loaded when you run `library(tidyverse)`. This means you must explicitly load it by running `library(lubridate)`.

`mondate`

This is a specialized package for handling dates in units of months in addition to days and years. It can be helpful in accounting and actuarial work, for example, where month-by-month calculations are needed.

`timeDate`

This is a high-powered package with well-thought-out facilities for handling dates and times, including date arithmetic, business days, holidays, conversions, and generalized handling of time zones. It was originally part of the Rmetrics software for financial modeling, where precision in dates and times is critical. If you have a demanding need for date facilities, consider this package.

Which class should you select? The article “[Date and Time Classes in R](#)” by Gabor Grothendieck and Thomas Petzoldt offers this general advice:

When considering which class to use, always choose the least complex class that will support the application. That is, use `Date` if possible, otherwise use `chron` and otherwise use the POSIX classes. Such a strategy will greatly reduce the potential for error and increase the reliability of your application.

See Also

See `help(DateTimeClasses)` for more details regarding the built-in facilities. See the June 2004 article “[Date and Time Classes in R](#)” by Gabor Grothendieck and Thomas Petzoldt for a great introduction to the date and time facilities. The June 2001 article “[Date-Time Classes](#)” by Brian Ripley and Kurt Hornik discusses the two POSIX classes in particular. Chapter 16, “[Dates and Times](#)”, from the book *R for Data Science* by Garrett Grolemund and Hadley Wickham (O'Reilly) provides a great introduction to `lubridate`.

7.1 Getting the Length of a String

Problem

You want to know the length of a string.

Solution

Use the `nchar` function, not the `length` function.

Discussion

The `nchar` function takes a string and returns the number of characters in the string:

```
nchar("Moe")
#> [1] 3
nchar("Curly")
#> [1] 5
```

If you apply `nchar` to a vector of strings, it returns the length of each string:

```
s <- c("Moe", "Larry", "Curly")
nchar(s)
#> [1] 3 5 5
```

You might think the `length` function returns the length of a string. Nope. It returns the length of a *vector*. When you apply the `length` function to a single string, R returns the value 1 because it views that string as a singleton vector—a vector with one element:

```
length("Moe")
#> [1] 1
length(c("Moe", "Larry", "Curly"))
#> [1] 3
```

7.2 Concatenating Strings

Problem

You want to join together two or more strings into one string.

Solution

Use the `paste` function.

Discussion

The `paste` function concatenates several strings together. In other words, it creates a new string by joining the given strings end to end:

```
paste("Everybody", "loves", "stats.")
#> [1] "Everybody loves stats."
```

By default, `paste` inserts a single space between pairs of strings, which is handy if that's what you want and annoying otherwise. The `sep` argument lets you specify a different separator. Use an empty string ("") to run the strings together without separation:

```
paste("Everybody", "loves", "stats.", sep = "-")
#> [1] "Everybody-loves-stats."
paste("Everybody", "loves", "stats.", sep = "")
#> [1] "Everybodylovesstats."
```

It's a common idiom to want to concatenate strings together with no separator at all. The function `paste0` makes this very convenient:

```
paste0("Everybody", "loves", "stats.")
#> [1] "Everybodylovesstats."
```

The function is very forgiving about nonstring arguments. It tries to convert them to strings using the `as.character` function silently behind the scenes:

```
paste("The square root of twice pi is approximately", sqrt(2 * pi))
#> [1] "The square root of twice pi is approximately 2.506628274631"
```

If one or more arguments are vectors of strings, `paste` will generate all combinations of the arguments (because of recycling):

```
stooges <- c("Moe", "Larry", "Curly")
paste(stooges, "loves", "stats.")
#> [1] "Moe loves stats."    "Larry loves stats." "Curly loves stats."
```

Sometimes you want to join even those combinations into one big string. The `collapse` parameter lets you define a top-level separator and instructs `paste` to concatenate the generated strings using that separator:

```
paste(stooges, "loves", "stats", collapse = ", and ")
#> [1] "Moe loves stats, and Larry loves stats, and Curly loves stats"
```

7.3 Extracting Substrings

Problem

You want to extract a portion of a string according to position.

Solution

Use `substr(string, start, end)` to extract the substring that begins at `start` and ends at `end`.

Discussion

The `substr` function takes a string, a starting point, and an ending point. It returns the substring between the starting and ending points:

```
substr("Statistics", 1, 4) # Extract first 4 characters
#> [1] "Stat"
substr("Statistics", 7, 10) # Extract last 4 characters
#> [1] "tics"
```

Just like many R functions, `substr` lets the first argument be a vector of strings. In that case, it applies itself to every string and returns a vector of substrings:

```
ss <- c("Moe", "Larry", "Curly")
substr(ss, 1, 3) # Extract first 3 characters of each string
#> [1] "Moe" "Lar" "Cur"
```

In fact, all the arguments can be vectors, in which case `substr` will treat them as parallel vectors. From each string, it extracts the substring delimited by the corresponding entries in the starting and ending points. This can facilitate some useful tricks. For example, the following code snippet extracts the last two characters from each string; each substring starts on the penultimate character of the original string and ends on the final character:

```
cities <- c("New York, NY", "Los Angeles, CA", "Peoria, IL")
substr(cities, nchar(cities) - 1, nchar(cities))
#> [1] "NY" "CA" "IL"
```

You can extend this trick into mind-numbing territory by exploiting the Recycling Rule, but we suggest you avoid the temptation.

7.4 Splitting a String According to a Delimiter

Problem

You want to split a string into substrings. The substrings are separated by a delimiter.

Solution

Use `strsplit`, which takes two arguments, the string and the delimiter of the substrings:

```
strsplit(string, delimiter)
```

The `delimiter` can be either a simple string or a regular expression.

Discussion

It is common for a string to contain multiple substrings separated by the same delimiter. One example is a filepath, whose components are separated by slashes (/):

```
path <- "/home/mike/data/trials.csv"
```

We can split that path into its components by using `strsplit` with a delimiter of /:

```
strsplit(path, "/")
#> [[1]]
#> [1] ""           "home"        "mike"        "data"        "trials.csv"
```

Notice that the first “component” is actually an empty string because nothing preceded the first slash.

Also notice that `strsplit` returns a list and that each element of the list is a vector of substrings. This two-level structure is necessary because the first argument can be a vector of strings. Each string is split into its substrings (a vector), and then those vectors are returned in a list.

If you are operating only on a single string, you can pop out the first element like this:

```
strsplit(path, "/")[[1]]
#> [1] ""           "home"        "mike"        "data"        "trials.csv"
```

This example splits three filepaths and returns a three-element list:

```
paths <- c(
  "/home/mike/data/trials.csv",
  "/home/mike/data/errors.csv",
  "/home/mike/corr/reject.doc"
)
strsplit(paths, "/")
#> [[1]]
#> [1] ""           "home"        "mike"        "data"        "trials.csv"
#>
```

```
#> [[2]]
#> [1] ""           "home"       "mike"       "data"       "errors.csv"
#>
#> [[3]]
#> [1] ""           "home"       "mike"       "corr"      "reject.doc"
```

The second argument of `strsplit` (the *delimiter* argument) is actually much more powerful than these examples indicate. It can be a regular expression, letting you match patterns far more complicated than a simple string. In fact, to turn off the regular expression feature (and its interpretation of special characters), you must include the `fixed=TRUE` argument.

See Also

To learn more about regular expressions in R, see the help page for `regexp`. See O'Reilly's *Mastering Regular Expressions*, by Jeffrey E.F. Friedl, to learn more about regular expressions in general.

7.5 Replacing Substrings

Problem

Within a string, you want to replace one substring with another.

Solution

Use `sub` to replace the first instance of a substring:

```
sub(old, new, string)
```

Use `gsub` to replace all instances of a substring:

```
gsub(old, new, string)
```

Discussion

The `sub` function finds the first instance of the *old* substring within *string* and replaces it with the *new* substring:

```
str <- "Curly is the smart one. Curly is funny, too."
sub("Curly", "Moe", str)
#> [1] "Moe is the smart one. Curly is funny, too."
```

`gsub` does the same thing, but it replaces *all* instances of the substring (a global replace), not just of the first instance:

```
gsub("Curly", "Moe", str)
#> [1] "Moe is the smart one. Moe is funny, too."
```

To remove a substring altogether, simply set the new substring to be empty:

```
sub(" and SAS", "", "For really tough problems, you need R and SAS.")  
#> [1] "For really tough problems, you need R."
```

The `old` argument can be a regular expression, which allows you to match patterns much more complicated than a simple string. This is actually assumed by default, so you must set the `fixed=TRUE` argument if you don't want `sub` and `gsub` to interpret `old` as a regular expression.

See Also

To learn more about regular expressions in R, see the help page for `regexp`. See *Mastering Regular Expressions* to learn more about regular expressions in general.

7.6 Generating All Pairwise Combinations of Strings

Problem

You have two sets of strings, and you want to generate all combinations from those two sets (their Cartesian product).

Solution

Use the `outer` and `paste` functions together to generate the matrix of all possible combinations:

```
m <- outer(strings1, strings2, paste, sep = "")
```

Discussion

The `outer` function is intended to form the outer product. However, it allows a third argument to replace simple multiplication with any function. In this recipe we replace multiplication with string concatenation (`paste`), and the result is all combinations of strings.

Suppose we have four test sites and three treatments:

```
locations <- c("NY", "LA", "CHI", "HOU")  
treatments <- c("T1", "T2", "T3")
```

We can apply `outer` and `paste` to generate all combinations of test sites and treatments like so:

```
outer(locations, treatments, paste, sep = "-")  
#>      [,1]    [,2]    [,3]  
#> [1,] "NY-T1"  "NY-T2"  "NY-T3"  
#> [2,] "LA-T1"  "LA-T2"  "LA-T3"  
#> [3,] "CHI-T1" "CHI-T2" "CHI-T3"  
#> [4,] "HOU-T1" "HOU-T2" "HOU-T3"
```

The fourth argument of `outer` is passed to `paste`. In this case, we passed `sep = "-"` in order to define a hyphen as the separator between the strings.

The result of `outer` is a matrix. If you want the combinations in a vector instead, flatten the matrix using the `as.vector` function.

In the special case where you are combining a set with itself and order does not matter, the result will be duplicate combinations:

```
outer(treatments, treatments, paste, sep = "-")
#>      [,1]     [,2]     [,3]
#> [1,] "T1-T1"  "T1-T2"  "T1-T3"
#> [2,] "T2-T1"  "T2-T2"  "T2-T3"
#> [3,] "T3-T1"  "T3-T2"  "T3-T3"
```

Or you can use `expand.grid` to get a pair of vectors representing all combinations:

```
expand.grid(treatments, treatments)
#>   Var1 Var2
#> 1   T1   T1
#> 2   T2   T1
#> 3   T3   T1
#> 4   T1   T2
#> 5   T2   T2
#> 6   T3   T2
#> 7   T1   T3
#> 8   T2   T3
#> 9   T3   T3
```

But suppose we want all *unique* pairwise combinations of treatments. We can eliminate the duplicates by removing the lower triangle (or upper triangle). The `lower.tri` function identifies that triangle, so inverting it identifies all elements *outside* the lower triangle:

```
m <- outer(treatments, treatments, paste, sep = "-")
m[!lower.tri(m)]
#> [1] "T1-T1"  "T1-T2"  "T2-T2"  "T1-T3"  "T2-T3"  "T3-T3"
```

See Also

See [Recipe 13.3](#) for using `paste` to generate combinations of strings. The [gtools package on CRAN](#) has the functions `combinations` and `permutation`, which may be of help with related tasks.

7.7 Getting the Current Date

Problem

You need to know today's date.

Solution

The `Sys.Date` function returns the current date:

```
Sys.Date()  
#> [1] "2019-05-13"
```

Discussion

The `Sys.Date` function returns a `Date` object. In the preceding example it seems to return a string because the result is printed inside double quotes. What really happens, however, is that `Sys.Date` returns a `Date` object and then R converts that object into a string for printing purposes. You can see this by checking the class of the result from `Sys.Date`:

```
class(Sys.Date())  
#> [1] "Date"
```

See Also

See [Recipe 7.9](#).

7.8 Converting a String into a Date

Problem

You have the string representation of a date, such as "2018-12-31", and you want to convert that into a `Date` object.

Solution

You can use `as.Date`, but you must know the format of the string. By default, `as.Date` assumes the string looks like *yyyy-mm-dd*. To handle other formats, you must specify the `format` parameter of `as.Date`. Use `format="%m/%d/%Y"` if the date is in American style, for instance.

Discussion

This example shows the default format assumed by `as.Date`, which is the ISO 8601 standard format of *yyyy-mm-dd*:

```
as.Date("2018-12-31")
#> [1] "2018-12-31"
```

The `as.Date` function returns a `Date` object that (as in the prior recipe) is being converted here back to a string for printing; this explains the double quotes around the output.

The string can be in other formats, but you must provide a `format` argument so that `as.Date` can interpret your string. See the help page for the `strftime` function for details about allowed formats.

Being simple Americans, we often mistakenly try to convert the usual American date format (*mm/dd/yyyy*) into a `Date` object, with these unhappy results:

```
as.Date("12/31/2018")
#> Error in charToDate(x): character string is not in a standard
#> unambiguous format
```

Here is the correct way to convert an American-style date:

```
as.Date("12/31/2018", format = "%m/%d/%Y")
#> [1] "2018-12-31"
```

Observe that the `Y` in the format string is capitalized to indicate a four-digit year. If you're using two-digit years, specify a lowercase `y`.

7.9 Converting a Date into a String

Problem

You want to convert a `Date` object into a character string, usually because you want to print the date.

Solution

Use either `format` or `as.character`:

```
format(Sys.Date())
#> [1] "2019-05-13"
as.character(Sys.Date())
#> [1] "2019-05-13"
```

Both functions allow a `format` argument that controls the formatting. Use `format="%m/%d/%Y"` to get American-style dates, for example:

```
format(Sys.Date(), format = "%m/%d/%Y")
#> [1] "05/13/2019"
```

Discussion

The `format` argument defines the appearance of the resulting string. Normal characters, such as a slash (/) or hyphen (-), are simply copied to the output string. Each two-letter combination of a percent sign (%) followed by another character has special meaning. Some common ones are:

`%b`
Abbreviated month name (“Jan”)

`%B`
Full month name (“January”)

`%d`
Day as a two-digit number

`%m`
Month as a two-digit number

`%y`
Year without century (00–99)

`%Y`
Year with century

See the help page for the `strftime` function for a complete list of formatting codes.

7.10 Converting Year, Month, and Day into a Date

Problem

You have a date represented by its year, month, and day in different variables. You want to merge these elements into a single `Date` object representation.

Solution

Use the `ISOdate` function:

```
ISOdate(year, month, day)
```

The result is a `POSIXct` object that you can convert into a `Date` object:

```
year <- 2018
month <- 12
day <- 31
```

```
as.Date(ISOdate(year, month, day))
#> [1] "2018-12-31"
```

Discussion

It is common for input data to contain dates encoded as three numbers: year, month, and day. The `ISOdate` function can combine them into a `POSIXct` object:

```
ISOdate(2020, 2, 29)
#> [1] "2020-02-29 12:00:00 GMT"
```

You can keep your date in the `POSIXct` format. However, when working with pure dates (not dates and times), we often convert to a `Date` object and truncate the unused time information:

```
as.Date(ISOdate(2020, 2, 29))
#> [1] "2020-02-29"
```

Trying to convert an invalid date results in NA:

```
ISOdate(2013, 2, 29) # Oops! 2013 is not a leap year
#> [1] NA
```

`ISOdate` can process entire vectors of years, months, and days, which is quite handy for mass conversion of input data. The following example starts with the year/month/day numbers for the third Wednesday in January of several years and then combines them all into `Date` objects:

```
years <- c(2010, 2011, 2012, 2014)
months <- c(1, 1, 1, 1, 1)
days <- c(15, 21, 20, 18, 17)
ISOdate(years, months, days)
#> [1] "2010-01-05 12:00:00 GMT" "2011-01-06 12:00:00 GMT"
#> [3] "2012-01-07 12:00:00 GMT" "2013-01-08 12:00:00 GMT"
#> [5] "2014-01-09 12:00:00 GMT"
as.Date(ISOdate(years, months, days))
#> [1] "2010-01-05" "2011-01-06" "2012-01-07" "2013-01-08" "2014-01-09"
```

Purists will note that the vector of months is redundant and that the last expression can therefore be further simplified by invoking the Recycling Rule:

```
as.Date(ISOdate(years, 1, days))
#> [1] "2010-01-05" "2011-01-06" "2012-01-07" "2013-01-08" "2014-01-09"
```

You can also extend this recipe to handle year, month, day, hour, minute, and second data by using the `ISOdatetime` function (see the help page for details):

```
ISOdatetime(year, month, day, hour, minute, second)
```

7.11 Getting the Julian Date

Problem

Given a `Date` object, you want to extract the Julian date—which is, in R, the number of days since January 1, 1970.

Solution

Either convert the `Date` object to an integer or use the `julian` function:

```
d <- as.Date("2019-03-15")
as.integer(d)
#> [1] 17970
jd <- julian(d)
jd
#> [1] 17970
#> attr(,"origin")
#> [1] "1970-01-01"
attr(jd, "origin")
#> [1] "1970-01-01"
```

Discussion

A Julian “date” is simply the number of days since an arbitrary starting point. In the case of R, that starting point is January 1, 1970, the same starting point as Unix systems. So the Julian date for January 1, 1970 is zero, as shown here:

```
as.integer(as.Date("1970-01-01"))
#> [1] 0
as.integer(as.Date("1970-01-02"))
#> [1] 1
as.integer(as.Date("1970-01-03"))
#> [1] 2
```

7.12 Extracting the Parts of a Date

Problem

Given a `Date` object, you want to extract a date part such as the day of the week, the day of the year, the calendar day, the calendar month, or the calendar year.

Solution

Convert the `Date` object to a `POSIXlt` object, which is a list of date parts. Then extract the desired part from that list:

```
d <- as.Date("2019-03-15")
p <- as.POSIXlt(d)
p$mday      # Day of the month
#> [1] 15
p$mon       # Month (0 = January)
#> [1] 2
p$year + 1900 # Year
#> [1] 2019
```

Discussion

The `POSIXlt` object represents a date as a list of date parts. Convert your `Date` object to `POSIXlt` by using the `as.POSIXlt` function, which will give you a list with these members:

```
sec          Seconds (0–61)
min          Minutes (0–59)
hour         Hours (0–23)
mday         Day of the month (1–31)
mon          Month (0–11)
year         Years since 1900
wday         Day of the week (0–6, 0 = Sunday)
yday         Day of the year (0–365)
isdst        Daylight Saving Time flag
```

Using these date parts, we can learn that April 2, 2020, is a Thursday (`wday` = 4) and the 93rd day of the year (because `yday` = 0 on January 1):

```
d <- as.Date("2020-04-02")
as.POSIXlt(d)$wday
#> [1] 4
as.POSIXlt(d)$yday
#> [1] 92
```

A common mistake is failing to add 1900 to the year, giving the impression you are living a long, long time ago:

```
as.POSIXlt(d)$year # Oops!  
#> [1] 120  
as.POSIXlt(d)$year + 1900  
#> [1] 2020
```

7.13 Creating a Sequence of Dates

Problem

You want to create a sequence of dates, such as a sequence of daily, monthly, or annual dates.

Solution

The `seq` function is a generic function that has a version for `Date` objects. It can create a `Date` sequence similarly to the way it creates a sequence of numbers.

Discussion

A typical use of `seq` specifies a starting date (`from`), ending date (`to`), and increment (`by`). An increment of 1 indicates daily dates:

```
s <- as.Date("2019-01-01")  
e <- as.Date("2019-02-01")  
seq(from = s, to = e, by = 1) # One month of dates  
#> [1] "2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"  
#> [6] "2019-01-06" "2019-01-07" "2019-01-08" "2019-01-09" "2019-01-10"  
#> [11] "2019-01-11" "2019-01-12" "2019-01-13" "2019-01-14" "2019-01-15"  
#> [16] "2019-01-16" "2019-01-17" "2019-01-18" "2019-01-19" "2019-01-20"  
#> [21] "2019-01-21" "2019-01-22" "2019-01-23" "2019-01-24" "2019-01-25"  
#> [26] "2019-01-26" "2019-01-27" "2019-01-28" "2019-01-29" "2019-01-30"  
#> [31] "2019-01-31" "2019-02-01"
```

Another typical use specifies a starting date (`from`), increment (`by`), and number of dates (`length.out`):

```
seq(from = s, by = 1, length.out = 7) # Dates, one week apart  
#> [1] "2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"  
#> [6] "2019-01-06" "2019-01-07"
```

The increment (`by`) is flexible and can be specified in days, weeks, months, or years:

```
seq(from = s, by = "month", length.out = 12) # First of the month for one year  
#> [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01" "2019-05-01"  
#> [6] "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01" "2019-10-01"  
#> [11] "2019-11-01" "2019-12-01"  
seq(from = s, by = "3 months", length.out = 4) # Quarterly dates for one year
```

```
#> [1] "2019-01-01" "2019-04-01" "2019-07-01" "2019-10-01"  
seq(from = s, by = "year", length.out = 10) # Year-start dates for one decade  
#> [1] "2019-01-01" "2020-01-01" "2021-01-01" "2022-01-01" "2023-01-01"  
#> [6] "2024-01-01" "2025-01-01" "2026-01-01" "2027-01-01" "2028-01-01"
```

Be careful with `by="month"` near month-end. In this example, the end of February overflows into March, which is probably not what you want:

```
seq(as.Date("2019-01-29"), by = "month", len = 3)  
#> [1] "2019-01-29" "2019-03-01" "2019-03-29"
```


CHAPTER 8

Probability

Probability theory is the foundation of statistics, and R has plenty of machinery for working with probability, probability distributions, and random variables. The recipes in this chapter show you how to calculate probabilities from quantiles, calculate quantiles from probabilities, generate random variables drawn from distributions, plot distributions, and so forth.

Names of Distributions

R has an abbreviated name for every probability distribution. This name is used to identify the functions associated with the distribution. For example, the name of the normal distribution is “norm,” which is the root of the function names listed in [Table 8-1](#).

Table 8-1. Normal distribution functions

Function	Purpose
<code>dnorm</code>	Normal density
<code>pnorm</code>	Normal distribution function
<code>qnorm</code>	Normal quantile function
<code>rnorm</code>	Normal random variates

[Table 8-2](#) describes some common discrete distributions, and [Table 8-3](#) describes several common continuous distributions.

Table 8-2. Common discrete distributions

Discrete distribution	R name	Parameters
Binomial	binom	n = number of trials; p = probability of success for one trial
Geometric	geom	p = probability of success for one trial
Hypergeometric	hyper	m = number of white balls in urn; n = number of black balls in urn; k = number of balls drawn from urn
Negative binomial (NegBinomial)	nbinom	size = number of successful trials; either prob = probability of successful trial or mu = mean
Poisson	pois	lambda = mean

Table 8-3. Common continuous distributions

Continuous distribution	R name	Parameters
Beta	beta	shape1; shape2
Cauchy	cauchy	location; scale
Chi-squared (Chisquare)	chisq	df = degrees of freedom
Exponential	exp	rate
F	f	df1 and df2 = degrees of freedom
Gamma	gamma	rate or scale
Log-normal (Lognormal)	lnorm	meanlog = mean on logarithmic scale; sdlog = standard deviation on logarithmic scale
Logistic	logis	location; scale
Normal	norm	mean; sd = standard deviation
Student's t (TDist)	t	df = degrees of freedom
Uniform	unif	min = lower limit; max = upper limit
Weibull	weibull	shape; scale
Wilcoxon	wilcox	m = number of observations in first sample; n = number of observations in second sample



All distribution-related functions require distributional parameters, such as size and prob for the binomial or prob for the geometric. The big “gotcha” is that the distributional parameters may not be what you expect. For example, we would expect the parameter of an exponential distribution to be β , the mean. The R convention, however, is for the exponential distribution to be defined by the rate = $1/\beta$, so we often supply the wrong value. The moral is, study the help page before you use a function related to a distribution. Be sure you’ve got the parameters right.

Getting Help on Probability Distributions

To see the R functions related to a particular probability distribution, use the `help` command and the full name of the distribution. For example, this will show the functions related to the normal distribution:

```
?Normal
```

Some distributions have names that don't work well with the `help` command, such as "Student's *t*." They have special help names, as noted in [Table 8-2](#) and [Table 8-3](#): NegBinomial, Chisquare, Lognormal, and TDist. Thus, to get help on the Student's *t* distribution, use this:

```
?TDist
```

See Also

There are many other distributions implemented in downloadable packages; see the CRAN task view devoted to [probability distributions](#). The `SuppDists` package is part of the R base, and it includes 10 supplemental distributions. The `MASS` package, which is also part of the base, provides additional support for distributions, such as maximum-likelihood fitting for some common distributions as well as sampling from a multivariate normal distribution.

8.1 Counting the Number of Combinations

Problem

You want to calculate the number of combinations of n items taken k at a time.

Solution

Use the `choose` function:

```
choose(n, k)
```

Discussion

A common problem in computing probabilities of discrete variables is counting combinations: the number of distinct subsets of size k that can be created from n items. The number is given by $n!/r!(n - r)!$, but it's much more convenient to use the `choose` function—especially as n and k grow larger:

```
choose(5, 3)  # How many ways can we select 3 items from 5 items?  
#> [1] 10  
choose(50, 3) # How many ways can we select 3 items from 50 items?  
#> [1] 19600
```

```
choose(50, 30) # How many ways can we select 30 items from 50 items?  
#> [1] 4.71e+13
```

These numbers are also known as *binomial coefficients*.

See Also

This recipe merely counts the combinations; see [Recipe 8.2](#) to actually generate them.

8.2 Generating Combinations

Problem

You want to generate all combinations of n items taken k at a time.

Solution

Use the `combn` function:

```
items <- 2:5  
k <- 2  
combn(items, k)  
#> [,1] [,2] [,3] [,4] [,5] [,6]  
#> [1,] 2 2 2 3 3 4  
#> [2,] 3 4 5 4 5 5
```

Discussion

We can use `combn(1:5, 3)` to generate all combinations of the numbers 1 through 5 taken three at a time:

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

The function is not restricted to numbers. We can generate combinations of strings, too. Here are all combinations of five treatments taken three at a time:

```
combn(c("T1", "T2", "T3", "T4", "T5"), 3)  
#> [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
#> [1,] "T1" "T1" "T1" "T1" "T1" "T1" "T2" "T2" "T2" "T3"  
#> [2,] "T2" "T2" "T2" "T3" "T3" "T4" "T3" "T4" "T4" "T4"  
#> [3,] "T3" "T4" "T5" "T4" "T5" "T5" "T4" "T5" "T5" "T5"
```



As the number of items, n , increases, the number of combinations can explode—especially if k is not near to 1 or n .

See Also

See [Recipe 8.1](#) to count the number of possible combinations *before* you generate a huge set.

8.3 Generating Random Numbers

Problem

You want to generate random numbers.

Solution

The simple case of generating a uniform random number between 0 and 1 is handled by the `runif` function. This example generates one uniform random number:

```
runif(1)
#> [1] 0.915
```



If you are saying `runif` out loud (or even in your head), you should pronounce it “are unif” instead of “run if.” The term `runif` is a *portmanteau* of “random uniform” so should not sound as if it’s a flow control function.

R can generate random variates from other distributions as well. For a given distribution, the name of the random number generator is “r” prefixed to the distribution’s abbreviated name (e.g., `rnorm` for the normal distribution’s random number generator). This example generates one random value from the standard normal distribution:

```
rnorm(1)
#> [1] 1.53
```

Discussion

Most programming languages have a wimpy random number generator that generates one random number, uniformly distributed between 0.0 and 1.0, and that’s all. Not R.

R can generate random numbers from many probability distributions other than the uniform distribution. The simple case of generating uniform random numbers between 0 and 1 is handled by the `runif` function:

```
runif(1)
#> [1] 0.83
```

The argument of `runif` is the number of random values to be generated. Generating a vector of 10 such values is as easy as generating one:

```
runif(10)
#> [1] 0.642 0.519 0.737 0.135 0.657 0.705 0.458 0.719 0.935 0.255
```

There are random number generators for all built-in distributions. Simply prefix the distribution name with “r” and you have the name of the corresponding random number generator. Here are some common ones:

```
runif(1, min = -3, max = 3)      # One uniform variate between -3 and +3
#> [1] 2.49
rnorm(1)                         # One standard Normal variate
#> [1] 1.53
rnorm(1, mean = 100, sd = 15)    # One Normal variate, mean 100 and SD 15
#> [1] 114
rbinom(1, size = 10, prob = 0.5) # One binomial variate
#> [1] 5
rpois(1, lambda = 10)           # One Poisson variate
#> [1] 12
rexp(1, rate = 0.1)              # One exponential variate
#> [1] 3.14
rgamma(1, shape = 2, rate = 0.1) # One gamma variate
#> [1] 22.3
```

As with `runif`, the first argument is the number of random values to be generated. Subsequent arguments are the parameters of the distribution, such as `mean` and `sd` for the normal distribution or `size` and `prob` for the binomial. See the function’s R help page for details.

The examples given so far use simple scalars for distributional parameters. Yet the parameters can also be vectors, in which case R will cycle through the vector while generating random values. The following example generates three normal random values drawn from distributions with means of -10 , 0 , and $+10$, respectively (all distributions have a standard deviation of 1.0):

```
rnorm(3, mean = c(-10, 0, +10), sd = 1)
#> [1] -9.420 -0.658 11.555
```

That is a powerful capability in cases such as hierarchical models, where the parameters are themselves random. The next example calculates 30 draws of a normal variate whose mean is itself randomly distributed and with hyperparameters of $\mu = 0$ and $\sigma = 0.2$:

```
means <- rnorm(30, mean = 0, sd = 0.2)
rnorm(30, mean = means, sd = 1)
#> [1] -0.5549 -2.9232 -1.2203  0.6962  0.1673 -1.0779 -0.3138 -3.3165
#> [9]  1.5952  0.8184 -0.1251  0.3601 -0.8142  0.1050  2.1264  0.6943
#> [17] -2.7771  0.9026  0.0389  0.2280 -0.5599  0.9572  0.1972  0.2602
#> [25] -0.4423  1.9707  0.4553  0.0467  1.5229  0.3176
```

If you are generating many random values and the vector of parameters is too short, R will apply the Recycling Rule to the parameter vector.

See Also

See the introduction to this chapter.

8.4 Generating Reproducible Random Numbers

Problem

You want to generate a sequence of random numbers, but you want to reproduce the same sequence every time your program runs.

Solution

Before running your R code, call the `set.seed` function to initialize the random number generator to a known state:

```
set.seed(42) # Or use any other positive integer...
```

Discussion

After generating random numbers, you may often want to reproduce the same sequence of “random” numbers every time your program executes. That way, you get the same results from run to run. One of the authors once supported a complicated Monte Carlo analysis of a huge portfolio of securities. The users complained about getting slightly different results each time the program ran. No kidding! The analysis was driven entirely by random numbers, so of course there was randomness in the output. The solution was to set the random number generator to a known state at the beginning of the program. That way, it would generate the same (quasi-)random numbers each time and thus yield consistent, reproducible results.

In R, the `set.seed` function sets the random number generator to a known state. The function takes one argument, an integer. Any positive integer will work, but you must use the same one in order to get the same initial state.

The function returns nothing. It works behind the scenes, initializing (or reinitializing) the random number generator. The key here is that using the same seed restarts the random number generator back at the same place:

```
set.seed(165) # Initialize generator to known state
runif(10) # Generate ten random numbers
#> [1] 0.116 0.450 0.996 0.611 0.616 0.426 0.666 0.168 0.788 0.442

set.seed(165) # Reinitialize to the same known state
runif(10) # Generate the same ten "random" numbers
#> [1] 0.116 0.450 0.996 0.611 0.616 0.426 0.666 0.168 0.788 0.442
```



When you set the seed value and freeze your sequence of random numbers, you are eliminating a source of randomness that may be critical to algorithms such as Monte Carlo simulations. Before you call `set.seed` in your application, ask yourself: am I undercutting the value of my program or perhaps even damaging its logic?

See Also

See [Recipe 8.3](#) for more about generating random numbers.

8.5 Generating a Random Sample

Problem

You want to sample a dataset randomly.

Solution

The `sample` function will randomly select n items from a set:

```
sample(set, n)
```

Discussion

Suppose your World Series data contains a vector of years when the Series was played. You can select 10 years at random using `sample`:

```
world_series <- read_csv("./data/world_series.csv")
sample(world_series$year, 10)
#> [1] 2010 1961 1906 1992 1982 1948 1910 1973 1967 1931
```

The items are randomly selected, so running `sample` again (usually) produces a different result:

```
sample(world_series$year, 10)
#> [1] 1941 1973 1921 1958 1979 1946 1932 1919 1971 1974
```

The `sample` function normally samples without replacement, meaning it will not select the same item twice. Some statistical procedures (especially the bootstrap)

require sampling *with* replacement, which means that one item can appear multiple times in the sample. Specify `replace=TRUE` to sample with replacement.

It's easy to implement a simple bootstrap using sampling with replacement. Suppose we have a vector, `x`, of 1,000 random numbers, drawn from a normal distribution with mean 4 and standard deviation 10:

```
set.seed(42)
x <- rnorm(1000, 4, 10)
```

This code fragment samples 1,000 times from `x` and calculates the median of each sample:

```
medians <- numeric(1000) # empty vector of 1000 numbers
for (i in 1:1000) {
  medians[i] <- median(sample(x, replace = TRUE))
}
```

From the bootstrap estimates, we can estimate the confidence interval for the median:

```
ci <- quantile(medians, c(0.025, 0.975))
cat("95% confidence interval is (", ci, ")\n")
#> 95% confidence interval is ( 3.16 4.49 )
```

We know that `x` was created from a normal distribution with a mean of 4, and hence the sample median should be 4 also. (In a symmetrical distribution like this one, the mean and the median are the same.) Our confidence interval easily contains the value.

See Also

See [Recipe 8.7](#) for randomly permuting a vector and [Recipe 13.8](#) for more about bootstrapping. [Recipe 8.4](#) discusses setting seeds for quasi-random numbers.

8.6 Generating Random Sequences

Problem

You want to generate a random sequence, such as a series of simulated coin tosses or a simulated sequence of Bernoulli trials.

Solution

Use the `sample` function. Sample `n` draws from the set of possible values, and set `replace=TRUE`:

```
sample(set, n, replace = TRUE)
```

Discussion

The `sample` function randomly selects items from a set. It normally samples *without replacement*, which means that it will not select the same item twice and will return an error if you try to sample more items than exist in the set. With `replace=TRUE`, however, `sample` can select items over and over; this allows you to generate long, random sequences of items.

The following example generates a random sequence of 10 simulated flips of a coin:

```
sample(c("H", "T"), 10, replace = TRUE)
#> [1] "H" "T" "H" "T" "T" "H" "T" "T" "H"
```

The next example generates a sequence of 20 Bernoulli trials—random successes or failures. We use `TRUE` to signify a success:

```
sample(c(FALSE, TRUE), 20, replace = TRUE)
#> [1] TRUE FALSE TRUE TRUE FALSE TRUE FALSE FALSE TRUE TRUE FALSE
#> [12] TRUE TRUE FALSE TRUE TRUE FALSE FALSE FALSE FALSE
```

By default `sample` will choose equally among the set elements, so the probability of selecting either `TRUE` or `FALSE` is 0.5. With a Bernoulli trial, the probability p of success is not necessarily 0.5. You can bias the sample by using the `prob` argument of `sample`; this argument is a vector of probabilities, one for each set element. Suppose we want to generate 20 Bernoulli trials with a probability of success $p = 0.8$. We set the probability of `FALSE` to be 0.2 and the probability of `TRUE` to 0.8:

```
sample(c(FALSE, TRUE), 20, replace = TRUE, prob = c(0.2, 0.8))
#> [1] TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
#> [12] TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE TRUE
```

The resulting sequence is clearly biased toward `TRUE`. We chose this example because it's a simple demonstration of a general technique. For the special case of a binary-valued sequence you can use `rbinom`, the random generator for binomial variates:

```
rbinom(10, 1, 0.8)
#> [1] 1 0 1 1 1 1 0 1 1
```

8.7 Randomly Permuting a Vector

Problem

You want to generate a random permutation of a vector.

Solution

If `v` is your vector, then `sample(v)` returns a random permutation.

Discussion

We typically think of the `sample` function for sampling from large datasets. However, the default parameters enable you to create a random rearrangement of the dataset. The function call `sample(v)` is equivalent to:

```
sample(v, size = length(v), replace = FALSE)
```

which means “select all the elements of `v` in random order while using each element exactly once.” That is a random permutation. Here is a random permutation of 1, ..., 10:

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

See Also

See [Recipe 8.5](#) for more about `sample`.

8.8 Calculating Probabilities for Discrete Distributions

Problem

You want to calculate either the simple or the cumulative probability associated with a discrete random variable.

Solution

For a simple probability, $P(X = x)$, use the density function. All built-in probability distributions have a density function whose name is “`d`” prefixed to the distribution name; for example, `dbinom` for the binomial distribution.

For a cumulative probability, $P(X \leq x)$, use the distribution function. All built-in probability distributions have a distribution function whose name is “`p`” prefixed to the distribution name; thus, `pbinom` is the distribution function for the binomial distribution.

Discussion

Suppose we have a binomial random variable X over 10 trials, where each trial has a success probability of 1/2. Then we can calculate the probability of observing $x = 7$ by calling `dbinom`:

```
dbinom(7, size = 10, prob = 0.5)
#> [1] 0.117
```

That calculates a probability of about 0.117. R calls `dbinom` the *density function*. Some textbooks call it the *probability mass function* or the *probability function*. Calling it a density function keeps the terminology consistent between discrete and continuous distributions (see [Recipe 8.9](#)).

The cumulative probability, $P(X \leq x)$, is given by the *distribution function*, which is sometimes called the *cumulative probability function*. The distribution function for the binomial distribution is `pbinom`. Here is the cumulative probability for $x = 7$ (i.e., $P(X \leq 7)$):

```
pbinom(7, size = 10, prob = 0.5)
#> [1] 0.945
```

It appears the probability of observing $X \leq 7$ is about 0.945.

The density functions and distribution functions for some common discrete distributions are shown in [Table 8-4](#).

Table 8-4. Discrete distributions

Distribution	Density function: $P(X=x)$	Distribution function: $P(X \leq x)$
Binomial	<code>dbinom(x, size, prob)</code>	<code>pbinom(x, size, prob)</code>
Geometric	<code>dgeom(x, prob)</code>	<code>pgeom(x, prob)</code>
Poisson	<code>dpois(x, lambda)</code>	<code>ppois(x, lambda)</code>

The complement of the cumulative probability is the *survival function*, $P(X > x)$. All of the distribution functions let you find this right-tail probability simply by specifying `lower.tail=FALSE`:

```
pbinom(7, size = 10, prob = 0.5, lower.tail = FALSE)
#> [1] 0.0547
```

Thus we see that the probability of observing $X > 7$ is about 0.055.

The *interval probability*, $P(x_1 < X \leq x_2)$, is the probability of observing X between the limits x_1 and x_2 . It is calculated as the difference between two cumulative probabilities: $P(X \leq x_2) - P(X \leq x_1)$. Here is $P(3 < X \leq 7)$ for our binomial variable:

```
pbinom(7, size = 10, prob = 0.5) - pbinom(3, size = 10, prob = 0.5)
#> [1] 0.773
```

R lets you specify multiple values of x for these functions and will return a vector of the corresponding probabilities. Here we calculate two cumulative probabilities, $P(X \leq 3)$ and $P(X \leq 7)$, in one call to `pbinom`:

```
pbinom(c(3, 7), size = 10, prob = 0.5)
#> [1] 0.172 0.945
```

This leads to a one-liner for calculating interval probabilities. The `diff` function calculates the difference between successive elements of a vector. We apply it to the output of `pbinom` to obtain the difference in cumulative probabilities—in other words, the interval probability:

```
diff(pbinom(c(3, 7), size = 10, prob = 0.5))
#> [1] 0.773
```

See Also

See this chapter's introduction for more about the built-in probability distributions.

8.9 Calculating Probabilities for Continuous Distributions

Problem

You want to calculate the distribution function (DF) or cumulative distribution function (CDF) for a continuous random variable.

Solution

Use the distribution function, which calculates $P(X \leq x)$. All built-in probability distributions have a distribution function whose name is “p” prefixed to the distribution’s abbreviated name—for instance, `pnorm` for the normal distribution.

For example, we can calculate the probability of a draw being from a random standard normal distribution being below 0.8 as follows:

```
pnorm(q = .8, mean = 0, sd = 1)
#> [1] 0.788
```

Discussion

The R functions for probability distributions follow a consistent pattern, so the solution to this recipe is essentially identical to the solution for discrete random variables (see [Recipe 8.8](#)). The significant difference is that continuous variables have no “probability” at a single point, $P(X = x)$. Instead, they have a “density” at a point.

Given that consistency, the discussion of distribution functions in [Recipe 8.8](#) is applicable here, too. [Table 8-5](#) gives the distribution functions for several continuous distributions.

Table 8-5. Continuous distributions

Distribution	Distribution function: $P(X \leq x)$
Normal	<code>pnorm(x, mean, sd)</code>
Student's <i>t</i>	<code>pt(x, df)</code>
Exponential	<code>pexp(x, rate)</code>
Gamma	<code>pgamma(x, shape, rate)</code>
Chi-squared (χ^2)	<code>pchisq(x, df)</code>

We can use `pnorm` to calculate the probability that a man is shorter than 66 inches, assuming that men's heights are normally distributed with a mean of 70 inches and a standard deviation of 3 inches. Mathematically speaking, we want $P(X \leq 66)$ given that $X \sim N(70, 3)$:

```
pnorm(66, mean = 70, sd = 3)
#> [1] 0.0912
```

Likewise, we can use `pexp` to calculate the probability that an exponential variable with a mean of 40 could be less than 20:

```
pexp(20, rate = 1 / 40)
#> [1] 0.393
```

Just as for discrete probabilities, the functions for continuous probabilities use `lower.tail=FALSE` to specify the survival function, $P(X > x)$. This call to `pexp` gives the probability that the same exponential variable could be greater than 50:

```
pexp(50, rate = 1 / 40, lower.tail = FALSE)
#> [1] 0.287
```

Also like discrete probabilities, the interval probability for a continuous variable, $P(x_1 < X < x_2)$, is computed as the difference between two cumulative probabilities, $P(X < x_2) - P(X < x_1)$. For the same exponential variable, here is $P(20 < X < 50)$, the probability that it could fall between 20 and 50:

```
pexp(50, rate = 1 / 40) - pexp(20, rate = 1 / 40)
#> [1] 0.32
```

See Also

See this chapter's introduction for more about the built-in probability distributions.

8.10 Converting Probabilities to Quantiles

Problem

Given a probability p and a distribution, you want to determine the corresponding quantile for p : the value x such that $P(X \leq x) = p$.

Solution

Every built-in distribution includes a quantile function that converts probabilities to quantiles. The function's name is “q” prefixed to the distribution name; thus, for instance, `qnorm` is the quantile function for the normal distribution.

The first argument of the quantile function is the probability. The remaining arguments are the distribution's parameters, such as `mean`, `shape`, or `rate`:

```
qnorm(0.05, mean = 100, sd = 15)
#> [1] 75.3
```

Discussion

A common example of computing quantiles is when we compute the limits of a confidence interval. If we want to know the 95% confidence interval ($\alpha = 0.05$) of a standard Normal variable, then we need the quantiles with probabilities of $\alpha/2 = 0.025$ and $(1 - \alpha)/2 = 0.975$:

```
qnorm(0.025)
#> [1] -1.96
qnorm(0.975)
#> [1] 1.96
```

In the true spirit of R, the first argument of the quantile functions can be a vector of probabilities, in which case we get a vector of quantiles. We can simplify this example into a one-liner:

```
qnorm(c(0.025, 0.975))
#> [1] -1.96 1.96
```

All the built-in probability distributions provide a quantile function. [Table 8-6](#) shows the quantile functions for some common discrete distributions.

Table 8-6. Discrete quantile distributions

Distribution	Quantile function
Binomial	<code>qbinom(p, size, prob)</code>
Geometric	<code>qgeom(p, prob)</code>
Poisson	<code>qpois(p, lambda)</code>

Table 8-7 shows the quantile functions for common continuous distributions.

Table 8-7. Continuous quantile distributions

Distribution	Quantile function
Normal	<code>qnorm(p, mean, sd)</code>
Student's <i>t</i>	<code>qt(p, df)</code>
Exponential	<code>qexp(p, rate)</code>
Gamma	<code>qgamma(p, shape, rate)</code> or <code>qgamma(p, shape, scale)</code>
Chi-squared (χ^2)	<code>qchisq(p, df)</code>

See Also

Determining the quantiles of a dataset is different from determining the quantiles of a distribution—see [Recipe 9.5](#).

8.11 Plotting a Density Function

Problem

You want to plot the density function of a probability distribution.

Solution

Define a vector `x` over the domain. Apply the distribution's density function to `x` and then plot the result. If `x` is a vector of points over the domain you care about plotting, you then calculate the density using one of the `d_____` density functions, like `dlnorm` for lognormal or `dnorm` for normal:

```
dens <- data.frame(x = x,
                     y = d_____ (x))
ggplot(dens, aes(x, y)) + geom_line()
```

Here is a specific example that plots the standard normal distribution for the interval -3 to $+3$:

```
library(ggplot2)

x <- seq(-3, +3, 0.1)
dens <- data.frame(x = x, y = dnorm(x))

ggplot(dens, aes(x, y)) + geom_line()
```

Figure 8-1 shows the smooth density function.

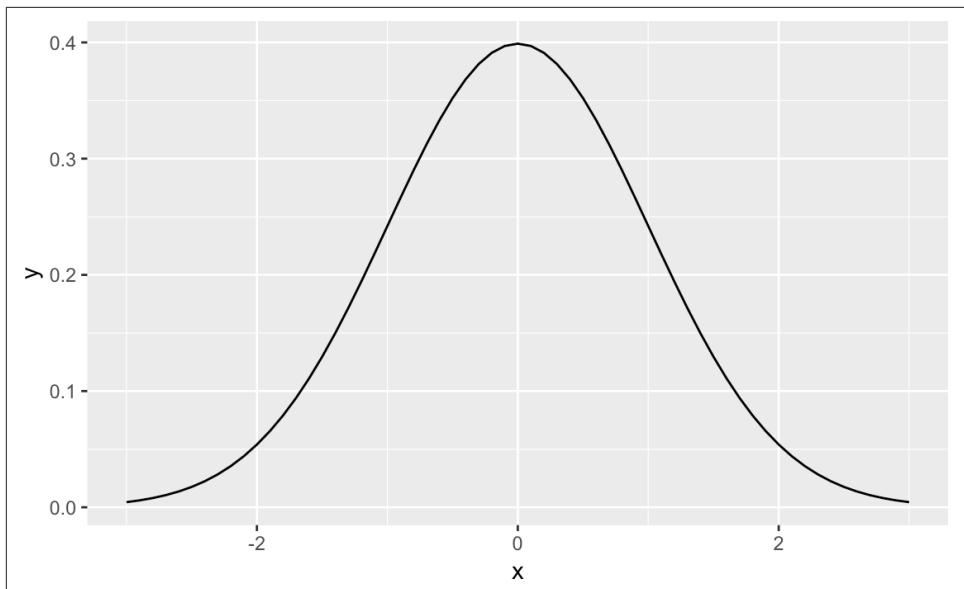


Figure 8-1. Smooth density function

Discussion

All the built-in probability distributions include a density function. For a particular density, the function name is “d” prepended to the distribution name. The density function for the normal distribution is `dnorm`, the density for the gamma distribution is `dgamma`, and so forth.

If the first argument of the density function is a vector, then the function calculates the density at each point and returns the vector of densities.

The following code creates a 2×2 plot of four densities (Figure 8-2):

```

x <- seq(from = 0, to = 6, length.out = 100) # Define the density domains
ylim <- c(0, 0.6)

# Make a data.frame with densities of several distributions
df <- rbind(
  data.frame(x = x, dist_name = "Uniform"=, y = dunif(x, min    = 2, max = 4)),
  data.frame(x = x, dist_name = "Normal"=, y = dnorm(x, mean   = 3, sd = 1)),
  data.frame(x = x, dist_name = "Exponential", y = dexp(x, rate  = 1 / 2)),
  data.frame(x = x, dist_name = "Gamma"=, y = dgamma(x, shape = 2, rate = 1)) )

# Make a line plot like before, but use facet_wrap to create the grid
ggplot(data = df, aes(x = x, y = y)) +
  geom_line() +
  facet_wrap(~dist_name)  # facet and wrap by the variable dist_name

```

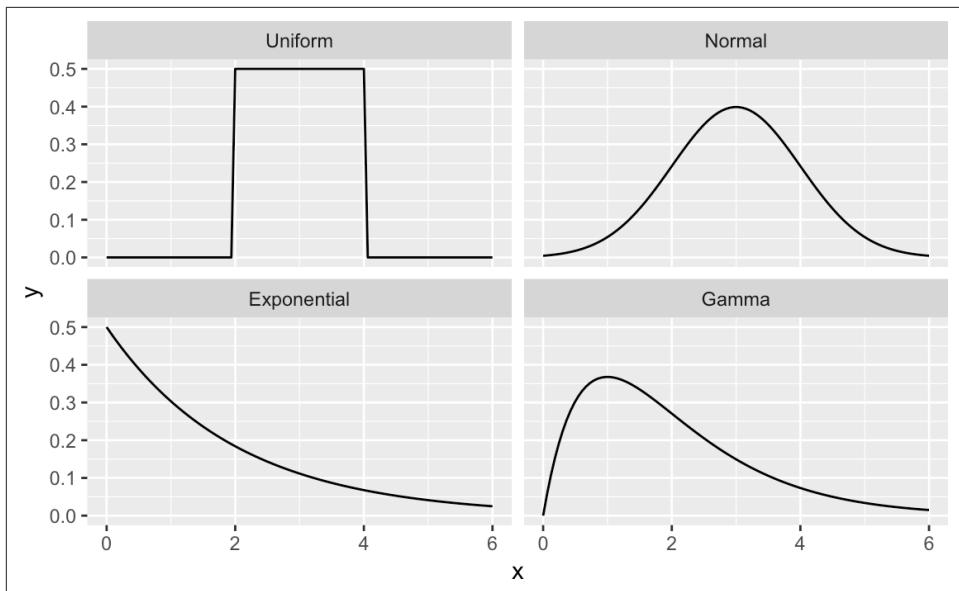


Figure 8-2. Multiple density plots

Figure 8-2 shows four density plots. However, a raw density plot is rarely useful or interesting by itself, and we often shade a region of interest.

Figure 8-3 is a normal distribution with shading from the 75th percentile to the 95th percentile.

We create the plot by plotting the density and then creating a shaded region with the `geom_ribbon` function from `ggplot2`.

First, we create some data and draw a density curve like the one shown in Figure 8-4:

```
x <- seq(from = -3, to = 3, length.out = 100)
df <- data.frame(x = x, y = dnorm(x, mean = 0, sd = 1))

p <- ggplot(df, aes(x, y)) +
  geom_line() +
  labs(
    title = "Standard Normal Distribution",
    y = "Density",
    x = "Quantile"
  )
p
```

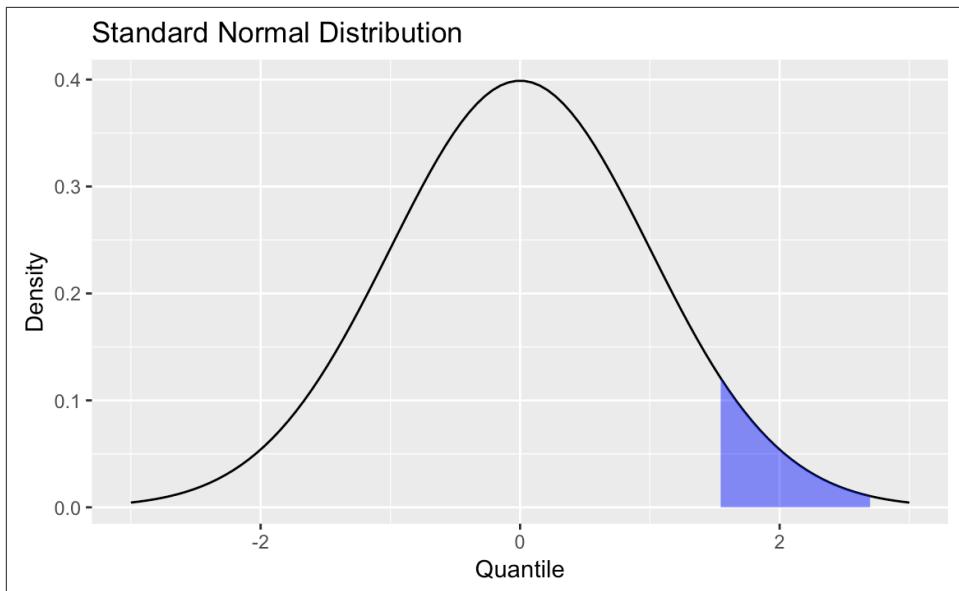


Figure 8-3. Standard normal with shading

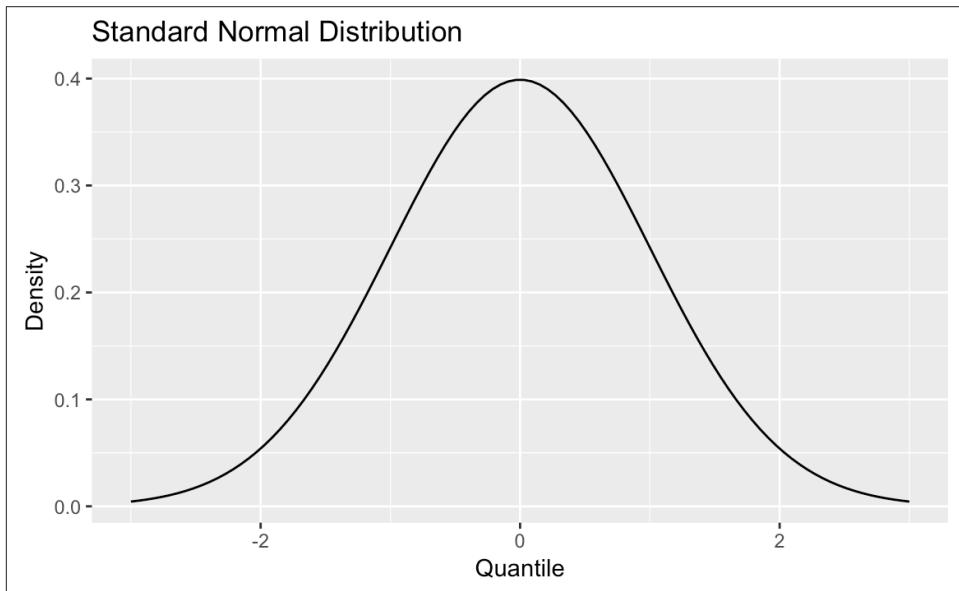


Figure 8-4. Density plot

Next, we define the region of interest by calculating the `x` values for the quantiles we're interested in. Finally, we use `geom_ribbon` to add a subset of our original data as a colored region:

```
q75 <- quantile(df$x, .75)
q95 <- quantile(df$x, .95)

p +
  geom_ribbon(
    data = subset(df, x > q75 & x < q95),
    aes(ymax = y),
    ymin = 0,
    fill = "blue",
    color = NA,
    alpha = 0.5
  )
```

The resulting plot is shown in [Figure 8-5](#).

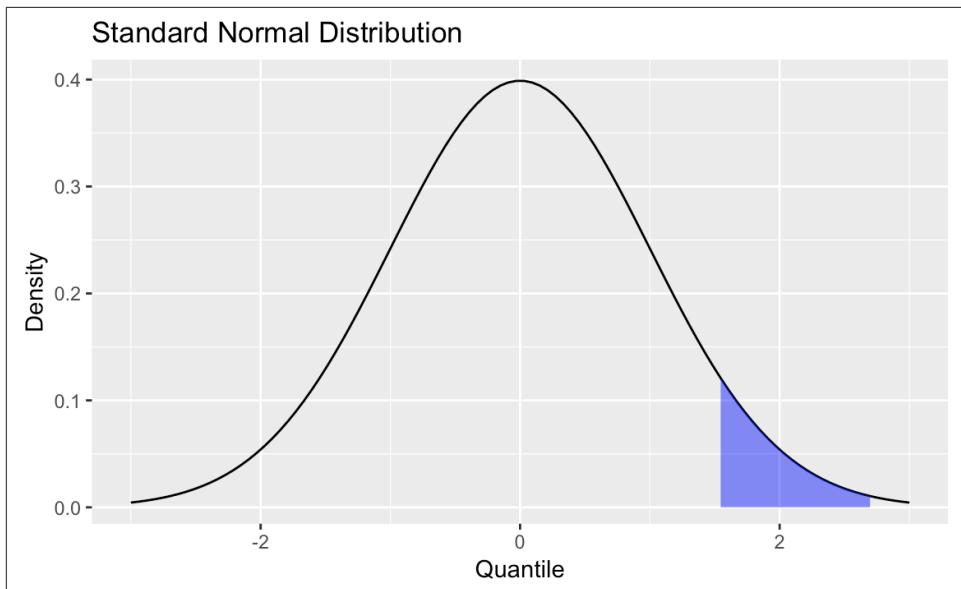


Figure 8-5. Normal density with shading

General Statistics

Any significant application of R includes statistics or models or graphics. This chapter addresses the statistics. Some recipes simply describe how to calculate a statistic, such as relative frequency. Most recipes involve statistical tests or confidence intervals. The statistical tests let you choose between two competing hypotheses; that paradigm is described next. Confidence intervals reflect the likely range of a population parameter and are calculated based on your data sample.

Null Hypotheses, Alternative Hypotheses, and p-Values

Many of the statistical tests in this chapter use a time-tested paradigm of statistical inference. In the paradigm, we have one or two data samples. We also have two competing hypotheses, either of which could reasonably be true.

One hypothesis, called the *null hypothesis*, is that *nothing happened*: the mean was unchanged; the treatment had no effect; you got the expected answer; the model did not improve; and so forth.

The other hypothesis, called the *alternative hypothesis*, is that *something happened*: the mean rose; the treatment improved the patients' health; you got an unexpected answer; the model fit better; and so forth.

We want to determine which hypothesis is more likely in light of the data. Here's how we do this:

1. To begin, we assume that the null hypothesis is true.
2. We calculate a test statistic. It could be something simple, such as the mean of the sample, or it could be quite complex. The critical requirement is that we must know the statistic's distribution. We might know the distribution of the sample mean, for example, by invoking the Central Limit Theorem.

3. From the statistic and its distribution we can calculate a p -value, the probability of a test statistic value as extreme or more extreme than the one we observed, while assuming that the null hypothesis is true.
4. If the p -value is too small, we have strong evidence against the null hypothesis. This is called *rejecting* the null hypothesis.
5. If the p -value is not small, then we have no such evidence. This is called *failing to reject* the null hypothesis.

There is one necessary decision here: when is a p -value “too small”?



In this book, we follow the common convention that we reject the null hypothesis when $p < 0.05$ and fail to reject it when $p > 0.05$. In statistical terminology, we choose a significance level of $\alpha = 0.05$ to define the border between strong evidence and insufficient evidence against the null hypothesis.

But the real answer is, “It depends.” Your chosen significance level depends on your problem domain. The conventional limit of $p < 0.05$ works for many problems. In our work, the data is especially noisy and so we are often satisfied with $p < 0.10$. For someone working in high-risk areas, $p < 0.01$ or $p < 0.001$ might be necessary.

In the recipes, we mention which tests include a p -value so that you can compare the p -value against your chosen significance level of α . We worded the recipes to help you interpret the comparison. Here is the wording from [Recipe 9.4](#), a test for the independence of two factors:

Conventionally, a p -value of less than 0.05 indicates that the variables are likely not independent, whereas a p -value exceeding 0.05 fails to provide any such evidence.

This is a compact way of saying:

- The null hypothesis is that the variables are independent.
- The alternative hypothesis is that the variables are not independent.
- For $\alpha = 0.05$, if $p < 0.05$ then we reject the null hypothesis, giving strong evidence that the variables are not independent; if $p > 0.05$, we fail to reject the null hypothesis.
- You are free to choose your own α , of course, in which case your decision to reject or fail to reject might be different.

Remember, the recipe states the *informal interpretation* of the test results, not the rigorous mathematical interpretation. We use colloquial language in the hope that it will guide you toward a practical understanding and application of the test. If the precise semantics of hypothesis testing are critical for your work, we urge you to consult the

reference cited under [See Also](#) or one of the other fine textbooks on mathematical statistics.

Confidence Intervals

Hypothesis testing is a well-understood mathematical procedure, but it can be frustrating. First, the semantics are tricky. The test does not reach a definite, useful conclusion. You might get strong evidence against the null hypothesis, but that's all you'll get. Second, it does not give you a number, only evidence.

If you want numbers then use confidence intervals, which bound the estimate of a population parameter at a given level of confidence. Recipes in this chapter can calculate confidence intervals for means, medians, and proportions of a population.

For example, [Recipe 9.9](#) calculates a 95% confidence interval for the population mean based on sample data. The interval is $97.16 < \mu < 103.98$, which means there is a 95% probability that the population's mean, μ , is between 97.16 and 103.98.

See Also

Statistical terminology and conventions can vary. This book generally follows the conventions of *Mathematical Statistics with Applications*, 6th ed., by Dennis Wackerly et al. (Duxbury Press). We recommend this book also for learning more about the statistical tests described in this chapter.

9.1 Summarizing Your Data

Problem

You want a basic statistical summary of your data.

Solution

The `summary` function gives some useful statistics for vectors, matrices, factors, and data frames:

```
summary(vec)
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#>     0.0    0.5    1.0    1.6    1.9   33.0
```

Discussion

The Solution exhibits the summary of a vector. The `1st Qu.` and `3rd Qu.` are the first and third quartile, respectively. Having both the median and mean is useful because you can quickly detect skew. The output in the Solution, for example, shows a mean

that is larger than the median; this indicates a possible skew to the right, as one would expect from a lognormal distribution.

The summary of a matrix works column by column. Here we see the summary of a matrix, `mat`, with three columns named `Samp1`, `Samp2`, and `Samp3`:

```
summary(mat)
#>      Samp1          Samp2          Samp3
#> Min.   : 1.0   Min.  :-2.943   Min.   : 0.04
#> 1st Qu.: 25.8  1st Qu.:-0.774  1st Qu.: 0.39
#> Median  : 50.5  Median :-0.052  Median  : 0.85
#> Mean    : 50.5  Mean   :-0.067  Mean   : 1.60
#> 3rd Qu.: 75.2  3rd Qu.: 0.684  3rd Qu.: 2.12
#> Max.    :100.0  Max.   : 2.150  Max.   :13.18
```

The summary of a factor gives counts:

```
summary(fac)
#> Maybe   No   Yes
#> 38     32   30
```

The summary of a character vector is pretty useless, giving just the vector length:

```
summary(char)
#> Length   Class    Mode
#>      100 character character
```

The summary of a data frame incorporates all these features. It works column by column, giving an appropriate summary according to the column type. Numeric values receive a statistical summary and factors are counted (character strings are not summarized):

```
suburbs <- read_csv("./data/suburbs.txt")
summary(suburbs)
#>      city          county          state
#> Length:17        Length:17        Length:17
#> Class :character  Class :character  Class :character
#> Mode  :character  Mode  :character  Mode  :character
#>
#>
#>
#>      pop
#> Min.   : 5428
#> 1st Qu.: 72616
#> Median  : 83048
#> Mean    : 249770
#> 3rd Qu.: 102746
#> Max.    :2853114
```

The “summary” of a list is pretty funky: you get the data type of each list member. Here is a `summary` of a list of vectors:

```
summary(vec_list)
#>   Length Class  Mode
#> x 100    -none- numeric
#> y 100    -none- numeric
#> z 100    -none- character
```

To summarize the data inside a list of vectors, map `summary` to each list element:

```
library(purrr)
map(vec_list, summary)
#> $x
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> -2.572 -0.686 -0.084 -0.043  0.660  2.413
#>
#> $y
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> -1.752 -0.589  0.045  0.079  0.769  2.293
#>
#> $z
#>   Length   Class   Mode
#>      100 character character
```

Unfortunately, the `summary` function does not compute any measure of variability, such as standard deviation or median absolute deviation. This is a serious shortcoming, so we usually call `sd` or `mad` (mean absolute deviation) right after calling `summary`.

See Also

See [Recipe 2.6](#) and [Recipe 6.1](#).

9.2 Calculating Relative Frequencies

Problem

You want to count the relative frequency of certain observations in your sample.

Solution

Identify the interesting observations by using a logical expression; then use the `mean` function to calculate the fraction of observations it identifies. For example, given a vector `x`, you can find the relative frequency of positive values in this way:

```
mean(x > 3)
#> [1] 0.12
```

Discussion

A logical expression, such as `x > 3`, produces a vector of logical values (TRUE and FALSE), one for each element of `x`. The `mean` function converts those values to 1s and

0s, respectively, and computes the average. This gives the fraction of values that are TRUE—in other words, the relative frequency of the interesting values. In the Solution, for example, that's the relative frequency of values greater than 3.

The concept here is pretty simple. The tricky part is dreaming up a suitable logical expression. Here are some examples:

```
mean(lab == "NJ")
Fraction of lab values that are New Jersey

mean(after > before)
Fraction of observations for which the effect increases

mean(abs(x-mean(x)) > 2*sd(x))
Fraction of observations that exceed two standard deviations from the mean

mean(diff(ts) > 0)
Fraction of observations in a time series that are larger than the previous obser-
vation
```

9.3 Tabulating Factors and Creating Contingency Tables

Problem

You want to tabulate one factor or build a contingency table from multiple factors.

Solution

The `table` function produces counts of one factor:

```
table(f1)
#> f1
#> a b c d e
#> 14 23 24 21 18
```

It can also produce contingency tables (cross-tabulations) from two or more factors:

```
table(f1, f2)
#> f2
#> f1   f   g   h
#> a    6   4   4
#> b    7   9   7
#> c    4  11   9
#> d    7   8   6
#> e    5  10   3
```

`table` works for characters, too, not only factors:

```
t1 <- sample(letters[9:11], 100, replace = TRUE)
table(t1)
```

```
#> t1  
#> i j k  
#> 20 40 40
```

Discussion

The `table` function counts the levels of one factor or characters, such as these counts of `initial` and `outcome` (which are factors):

```
set.seed(42)  
initial <- factor(sample(c("Yes", "No", "Maybe"), 100, replace = TRUE))  
outcome <- factor(sample(c("Pass", "Fail"), 100, replace = TRUE))  
  
table(initial)  
#> initial  
#> Maybe No Yes  
#> 39 31 30  
  
table(outcome)  
#> outcome  
#> Fail Pass  
#> 56 44
```

The greater power of `table` is in producing contingency tables, also known as cross-tabulations. Each cell in a contingency table counts how many times that row/column combination occurred:

```
table(initial, outcome)  
#>          outcome  
#> initial Fail Pass  
#>   Maybe   23   16  
#>   No     20   11  
#>   Yes    13   17
```

This table shows that the combination of `initial` = `Yes` and `outcome` = `Fail` occurred 13 times, the combination of `initial` = `Yes` and `outcome` = `Pass` occurred 17 times, and so forth.

See Also

The `xtabs` function can also produce a contingency table. It has a formula interface, which some people prefer.

9.4 Testing Categorical Variables for Independence

Problem

You have two categorical variables that are represented by factors. You want to test them for independence using the chi-squared test.

Solution

Use the `table` function to produce a contingency table from the two factors. Then use the `summary` function to perform a chi-squared test of the contingency table. In this example we have two vectors of factor values, which we created in the prior recipe:

```
summary(table(initial, outcome))
#> Number of cases in table: 100
#> Number of factors: 2
#> Test for independence of all factors:
#> Chisq = 3, df = 2, p-value = 0.2
```

The output includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the variables are likely not independent, whereas a p -value exceeding 0.05 fails to provide any such evidence.

Discussion

This example performs a chi-squared test on the contingency table from [Recipe 9.3](#), and yields a p -value of 0.2:

```
summary(table(initial, outcome))
#> Number of cases in table: 100
#> Number of factors: 2
#> Test for independence of all factors:
#> Chisq = 3, df = 2, p-value = 0.2
```

The large p -value indicates that the two factors, `initial` and `outcome`, are probably independent. Practically speaking, we conclude there is no connection between the variables. This makes sense, as this example data was created by simply drawing random data using the `sample` function in the prior recipe.

See Also

The `chisq.test` function can also perform this test.

9.5 Calculating Quantiles (and Quartiles) of a Dataset

Problem

Given a fraction f , you want to know the corresponding quantile of your data. That is, you seek the observation x such that the fraction of observations below x is f .

Solution

Use the `quantile` function. The second argument is the fraction, f .

```
quantile(vec, 0.95)
#>   95%
#> 1.43
```

For quartiles, simply omit the second argument altogether:

```
quantile(vec)
#>    0%    25%    50%    75%   100%
#> -2.0247 -0.5915 -0.0693  0.4618  2.7019
```

Discussion

Suppose `vec` contains 1,000 observations between 0 and 1. The `quantile` function can tell you which observation delimits the lower 5% of the data:

```
vec <- runif(1000)
quantile(vec, .05)
#>   5%
#> 0.0451
```

The `quantile` documentation refers to the second argument as a “probability,” which is natural when we think of probability as meaning relative frequency.

In true R style, the second argument can be a vector of probabilities; in this case, `quantile` returns a vector of corresponding quantiles, one for each probability:

```
quantile(vec, c(.05, .95))
#>   5%   95%
#> 0.0451 0.9363
```

That is a handy way to identify the middle 90% (in this case) of the observations.

If you omit the probabilities altogether, then R assumes you want the probabilities 0, 0.25, 0.50, 0.75, and 1.0—in other words, the quartiles:

```
quantile(vec)
#>    0%    25%    50%    75%   100%
#> 0.000405 0.235529 0.479543 0.737619 0.999379
```

Amazingly, the `quantile` function implements nine (yes, nine) different algorithms for computing quantiles. Study the help page before assuming that the default algorithm is the best one for you.

9.6 Inverting a Quantile

Problem

Given an observation x from your data, you want to know its corresponding quantile. That is, you want to know what fraction of the data is less than x .

Solution

Assuming your data is in a vector `vec`, compare the data against the observation and then use `mean` to compute the relative frequency of values less than x —say, 1.6 as per this example:

```
mean(vec < 1.6)
#> [1] 0.948
```

Discussion

The expression `vec < x` compares every element of `vec` against x and returns a vector of logical values, where the n th logical value is TRUE if `vec[n] < x`. The `mean` function converts those logical values to 0s and 1s: 0 for FALSE and 1 for TRUE. The average of all those 1s and 0s is the fraction of `vec` that is less than x , or the inverse quantile of x .

See Also

This is an application of the general approach described in [Recipe 9.2](#).

9.7 Converting Data to z-Scores

Problem

You have a dataset, and you want to calculate the corresponding z -scores for all data elements. (This is sometimes called *normalizing* the data.)

Solution

Use the `scale` function:

```
scale(x)
#>      [,1]
#> [1,]  0.8701
#> [2,] -0.7133
#> [3,] -1.0503
#> [4,]  0.5790
#> [5,] -0.6324
#> [6,]  0.0991
#> [7,]  2.1495
#> [8,]  0.2481
#> [9,] -0.8155
#> [10,] -0.7341
#> attr(,"scaled:center")
#> [1] 2.42
#> attr(,"scaled:scale")
#> [1] 2.11
```

This works for vectors, matrices, and data frames. In the case of a vector, `scale` returns the vector of normalized values. In the case of matrices and data frames, `scale` normalizes each column independently and returns columns of normalized values in a matrix.

Discussion

You might also want to normalize a single value y relative to a dataset x . You can do so by using vectorized operations as follows:

```
(y - mean(x)) / sd(x)  
#> [1] -0.633
```

9.8 Testing the Mean of a Sample (t-Test)

Problem

You have a sample from a population. Given this sample, you want to know if the mean of the population could reasonably be a particular value m .

Solution

Apply the `t.test` function to the sample x with the argument `mu = m`:

```
t.test(x, mu = m)
```

The output includes a p -value. Conventionally, if $p < 0.05$ then the population mean is unlikely to be m , whereas $p > 0.05$ provides no such evidence.

If your sample size n is small, then the underlying population must be normally distributed in order to derive meaningful results from the t -test. A good rule of thumb is that “small” means $n < 30$.

Discussion

The t -test is a workhorse of statistics, and this is one of its basic uses: making inferences about a population mean from a sample. The following example simulates sampling from a normal population with mean $\mu = 100$. It uses the t -test to ask if the population mean could be 95, and `t.test` reports a p -value of 0.005:

```
x <- rnorm(75, mean = 100, sd = 15)  
t.test(x, mu = 95)  
#>  
#> One Sample t-test  
#>  
#> data: x  
#> t = 3, df = 70, p-value = 0.005  
#> alternative hypothesis: true mean is not equal to 95
```

```
#> 95 percent confidence interval:  
#> 96.5 103.0  
#> sample estimates:  
#> mean of x  
#> 99.7
```

The p -value is small, so it's unlikely (based on the sample data) that 95 could be the mean of the population.

Informally, we could interpret the low p -value as follows. If the population mean were really 95, then the probability of observing our test statistic ($t = 2.8898$ or something more extreme) would be only 0.005. That is very improbable, yet that is the value we observed. Hence we conclude that the null hypothesis is wrong; therefore, the sample data does not support the claim that the population mean is 95.

In sharp contrast, testing for a mean of 100 gives a p -value of 0.9:

```
t.test(x, mu = 100)  
#>  
#> One Sample t-test  
#>  
#> data: x  
#> t = -0.2, df = 70, p-value = 0.9  
#> alternative hypothesis: true mean is not equal to 100  
#> 95 percent confidence interval:  
#> 96.5 103.0  
#> sample estimates:  
#> mean of x  
#> 99.7
```

The large p -value indicates that the sample is consistent with assuming a population mean μ of 100. In statistical terms, the data does not provide evidence against the true mean being 100.

A common case is testing for a mean of zero. If you omit the `mu` argument, it defaults to 0.

See Also

The `t.test` function is a many-splendored thing. See [Recipe 9.9](#) and [Recipe 9.15](#) for other uses.

9.9 Forming a Confidence Interval for a Mean

Problem

You have a sample from a population. Given that sample, you want to determine a confidence interval for the population's mean.

Solution

Apply the `t.test` function to your sample `x`:

```
t.test(x)
```

The output includes a confidence interval at the 95% confidence level. To see intervals at other levels, use the `conf.level` argument.

As in [Recipe 9.8](#), if your sample size n is small, then the underlying population must be normally distributed for there to be a meaningful confidence interval. Again, a good rule of thumb is that “small” means $n < 30$.

Discussion

Applying the `t.test` function to a vector yields a lot of output. Buried in the output is a confidence interval:

```
t.test(x)
#>
#> One Sample t-test
#>
#> data: x
#> t = 50, df = 50, p-value <2e-16
#> alternative hypothesis: true mean is not equal to 0
#> 95 percent confidence interval:
#> 94.2 101.5
#> sample estimates:
#> mean of x
#> 97.9
```

In this example, the confidence interval is approximately $94.2 < \mu < 101.5$, which is sometimes written simply as $(94.2, 101.5)$.

We can raise the confidence level to 99% by setting `conf.level=0.99`:

```
t.test(x, conf.level = 0.99)
#>
#> One Sample t-test
#>
#> data: x
#> t = 50, df = 50, p-value <2e-16
#> alternative hypothesis: true mean is not equal to 0
#> 99 percent confidence interval:
#> 92.9 102.8
#> sample estimates:
#> mean of x
#> 97.9
```

That change widens the confidence interval to $92.9 < \mu < 102.8$.

9.10 Forming a Confidence Interval for a Median

Problem

You have a data sample, and you want to know the confidence interval for the median.

Solution

Use the `wilcox.test` function, setting `conf.int=TRUE`:

```
wilcox.test(x, conf.int = TRUE)
```

The output will contain a confidence interval for the median.

Discussion

The procedure for calculating the confidence interval of a mean is well defined and widely known. The same is not true for the median, unfortunately. There are several procedures for calculating the median's confidence interval. None of them is "the" procedure, but the Wilcoxon signed rank test is pretty standard.

The `wilcox.test` function implements that procedure. Buried in the output is the 95% confidence interval, which is approximately (-0.102, 0.646) in this case:

```
wilcox.test(x, conf.int = TRUE)
#>
#> Wilcoxon signed rank test
#>
#> data: x
#> V = 200, p-value = 0.1
#> alternative hypothesis: true location is not equal to 0
#> 95 percent confidence interval:
#> -0.102 0.646
#> sample estimates:
#> (pseudo)median
#>                 0.311
```

You can change the confidence level by setting `conf.level`, such as `conf.level=0.99` or other such values.

The output also includes something called the *pseudomedian*, which is defined on the help page. Don't assume it equals the median; they are different:

```
median(x)
#> [1] 0.314
```

See Also

The bootstrap procedure is also useful for estimating the median's confidence interval; see [Recipe 8.5](#) and [Recipe 13.8](#).

9.11 Testing a Sample Proportion

Problem

You have a sample of values from a population consisting of successes and failures. You believe the true proportion of successes is p , and you want to test that hypothesis using the sample data.

Solution

Use the `prop.test` function. Suppose the sample size is n and the sample contains x successes:

```
prop.test(x, n, p)
```

The output includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the true proportion is unlikely to be p , whereas a p -value exceeding 0.05 fails to provide such evidence.

Discussion

Suppose you encounter some loudmouthed fan of the Chicago Cubs early in the baseball season. The Cubs have played 20 games and won 11 of them, or 55% of their games. Based on that evidence, the fan is “very confident” that the Cubs will win more than half of their games this year. Should they be that confident?

The `prop.test` function can evaluate the fan’s logic. Here, the number of observations is $n = 20$, the number of successes is $x = 11$, and p is the true probability of winning a game. We want to know whether it is reasonable to conclude, based on the data, that $p > 0.5$. Normally, `prop.test` would check for $p \neq 0.05$, but we can check for $p > 0.5$ instead by setting `alternative="greater"`:

```
prop.test(11, 20, 0.5, alternative = "greater")
#>
#> 1-sample proportions test with continuity correction
#>
#> data: 11 out of 20, null probability 0.5
#> X-squared = 0.05, df = 1, p-value = 0.4
#> alternative hypothesis: true p is greater than 0.5
#> 95 percent confidence interval:
#> 0.35 1.00
#> sample estimates:
```

```
#>      P  
#> 0.55
```

The `prop.test` output shows a large p -value, 0.55, so we cannot reject the null hypothesis; that is, we cannot reasonably conclude that p is greater than 1/2. The Cubs fan is being overly confident based on too little data. No surprise there.

9.12 Forming a Confidence Interval for a Proportion

Problem

You have a sample of values from a population consisting of successes and failures. Based on the sample data, you want to form a confidence interval for the population's proportion of successes.

Solution

Use the `prop.test` function. Suppose the sample size is n and the sample contains x successes:

```
prop.test(x, n)
```

The function output includes the confidence interval for p .

Discussion

We subscribe to a stock market newsletter that is well written, but includes a section purporting to identify stocks that are likely to rise. It does this by looking for a certain pattern in the stock price. It recently reported, for example, that a certain stock was following the pattern. It also reported that the stock rose six times after the last nine times that pattern occurred. The writers concluded that the probability of the stock rising again was therefore 6/9, or 66.7%.

Using `prop.test`, we can obtain the confidence interval for the true proportion of times the stock rises after the pattern. Here, the number of observations is $n = 9$ and the number of successes is $x = 6$. The output shows a confidence interval of (0.309, 0.910) at the 95% confidence level:

```
prop.test(6, 9)  
#> Warning in prop.test(6, 9): Chi-squared approximation may be incorrect  
#>  
#> 1-sample proportions test with continuity correction  
#>  
#> data: 6 out of 9, null probability 0.5  
#> X-squared = 0.4, df = 1, p-value = 0.5  
#> alternative hypothesis: true p is not equal to 0.5  
#> 95 percent confidence interval:  
#> 0.309 0.910
```

```
#> sample estimates:  
#>      p  
#> 0.667
```

The writers are pretty foolish to say the probability of the stock rising is 66.7%. They could be leading their readers into a very bad bet.

By default, `prop.test` calculates a confidence interval at the 95% confidence level. Use the `conf.level` argument for other confidence levels:

```
prop.test(x, n, p, conf.level = 0.99) # 99% confidence level
```

See Also

See [Recipe 9.11](#).

9.13 Testing for Normality

Problem

You want a statistical test to determine whether your data sample is from a normally distributed population.

Solution

Use the `shapiro.test` function:

```
shapiro.test(x)
```

The output includes a p -value. Conventionally, $p < 0.05$ indicates that the population is likely not normally distributed, whereas $p > 0.05$ provides no such evidence.

Discussion

This example reports a p -value of 0.4 for x :

```
shapiro.test(x)  
#>  
#> Shapiro-Wilk normality test  
#>  
#> data: x  
#> W = 1, p-value = 0.4
```

The large p -value suggests the underlying population could be normally distributed. The next example reports a very small p -value for y , so it is unlikely that this sample came from a normal population:

```
shapiro.test(y)  
#>  
#> Shapiro-Wilk normality test
```

```
#>  
#> data: y  
#> W = 0.7, p-value = 7e-13
```

We have highlighted the Shapiro–Wilk test because it is a standard R function. You can also install the package `nor.test`, which is dedicated entirely to tests for normality. This package includes the following tests:

- Anderson–Darling (`ad.test`)
- Cramer–von Mises (`cvm.test`)
- Lilliefors (`lillie.test`)
- Pearson chi-squared for the composite hypothesis of normality (`pearson.test`)
- Shapiro–Francia (`sf.test`)

The problem with all of these is their null hypothesis: they all assume that the population is normally distributed until proven otherwise. As a result, the population must be decidedly nonnormal before the test reports a small p -value and you can reject that null hypothesis. That makes the tests quite conservative, tending to err on the side of normality.

Instead of depending solely upon a statistical test, we suggest also using histograms (Recipe 10.19) and quantile-quantile plots (Recipe 10.21) to evaluate the normality of any data. Are the tails too fat? Is the peak too peaked? Your judgment is likely better than a single statistical test.

See Also

See Recipe 3.10 for how to install the `nor.test` package.

9.14 Testing for Runs

Problem

Your data is a sequence of binary values: yes/no, 0/1, true/false, or other two-valued data. You want to know: is the sequence random?

Solution

The `tseries` package contains the `runs.test` function, which checks a sequence for randomness. The sequence should be a factor with two levels:

```
library(tseries)  
runs.test(as.factor(s))
```

The `runs.test` function reports a p -value. Conventionally, a p -value of less than 0.05 indicates that the sequence is likely not random, whereas a p -value exceeding 0.05 provides no such evidence.

Discussion

A run is a subsequence composed of identical values, such as all 1s or all 0s. A random sequence should be properly jumbled up, without too many runs. It shouldn't contain too *few* runs, either—a sequence of perfectly alternating values (0, 1, 0, 1, 0, 1, ...) contains no runs, but would you say that it's random?

The `runs.test` function checks the number of runs in your sequence. If there are too many or too few, it reports a small p -value.

This first example generates a random sequence of 0s and 1s and then tests the sequence for runs. Not surprisingly, `runs.test` reports a large p -value, indicating the sequence is likely random:

```
s <- sample(c(0, 1), 100, replace = T)
runs.test(as.factor(s))
#>
#> Runs Test
#>
#> data: as.factor(s)
#> Standard Normal = 0.1, p-value = 0.9
#> alternative hypothesis: two.sided
```

This next sequence, however, consists of three runs and so the reported p -value is quite low:

```
s <- c(0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0)
runs.test(as.factor(s))
#>
#> Runs Test
#>
#> data: as.factor(s)
#> Standard Normal = -2, p-value = 0.02
#> alternative hypothesis: two.sided
```

See Also

See [Recipe 5.4](#) and [Recipe 8.6](#).

9.15 Comparing the Means of Two Samples

Problem

You have one sample each from two populations. You want to know if the two populations could have the same mean.

Solution

Perform a *t*-test by calling the `t.test` function:

```
t.test(x, y)
```

By default, `t.test` assumes that your observations are not paired. If the observations are paired (i.e., if each x_i is paired with one y_i), then specify `paired=TRUE`:

```
t.test(x, y, paired = TRUE)
```

In either case, `t.test` will compute a *p*-value. Conventionally, if $p < 0.05$ then the means are likely different, whereas $p > 0.05$ provides no such evidence:

- If either sample size is small, then the populations must be normally distributed. Here, “small” means fewer than 20 data points.
- If the two populations have the same variance, specify `var.equal=TRUE` to obtain a less conservative test.

Discussion

We often use the *t*-test to get a quick sense of the difference between two population means. It requires that the samples be large enough (i.e., both samples have 20 or more observations) or that the underlying populations be normally distributed. We don’t take the “normally distributed” part too literally. Being bell-shaped and reasonably symmetrical should be good enough.

A key distinction here is whether or not your data contains paired observations, since the results may differ in the two cases. Suppose we want to know if drinking coffee in the morning improves scores on SATs. We could run the experiment two ways:

- Randomly select one group of people. Give them the SAT twice, once with morning coffee and once without morning coffee. For each person, we will have two SAT scores. These are paired observations.
- Randomly select two groups of people. One group has a cup of morning coffee and takes the SAT. The other group just takes the test. We have a score for each person, but the scores are not paired in any way.

Statistically, these experiments are quite different. In experiment 1, there are two observations for each person (caffeinated and not) and they are not statistically independent. In experiment 2, the observations are independent.

If you have paired observations (experiment 1) and erroneously analyze them as unpaired observations (experiment 2), then you could get this result with a p -value of 0.3:

```
load("./data/sat.rdata")
t.test(x, y)
#>
#> Welch Two Sample t-test
#>
#> data: x and y
#> t = -1, df = 200, p-value = 0.3
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -46.4 16.2
#> sample estimates:
#> mean of x mean of y
#> 1054 1069
```

The large p -value forces you to conclude there is no difference between the groups. Contrast that result with the one that follows from analyzing the same data but correctly identifying it as paired:

```
t.test(x, y, paired = TRUE)
#>
#> Paired t-test
#>
#> data: x and y
#> t = -20, df = 100, p-value <2e-16
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -16.8 -13.5
#> sample estimates:
#> mean of the differences
#> -15.1
```

The p -value plummets to $2e-16$, and we reach the exactly opposite conclusion.

See Also

If the populations are not normally distributed (bell-shaped) and either sample is small, consider using the Wilcoxon–Mann–Whitney test described in [Recipe 9.16](#).

9.16 Comparing the Locations of Two Samples Nonparametrically

Problem

You have samples from two populations. You don't know the distribution of the populations, but you know they have similar shapes. You want to know: is one population shifted to the left or right compared with the other?

Solution

You can use a nonparametric test, the Wilcoxon–Mann–Whitney test, which is implemented by the `wilcox.test` function. For paired observations (every x_i is paired with y_i), set `paired=TRUE`:

```
wilcox.test(x, y, paired = TRUE)
```

For unpaired observations, let `paired` default to FALSE:

```
wilcox.test(x, y)
```

The test output includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the second population is likely shifted left or right with respect to the first population, whereas a p -value exceeding 0.05 provides no such evidence.

Discussion

When we stop making assumptions regarding the distributions of populations, we enter the world of nonparametric statistics. The Wilcoxon–Mann–Whitney test is nonparametric and so can be applied to more datasets than the t -test, which requires that the data be normally distributed (for small samples). This test's only assumption is that the two populations have the same shape.

In this recipe, we are asking: is the second population shifted left or right with respect to the first? This is similar to asking whether the average of the second population is smaller or larger than that of the first. However, the Wilcoxon–Mann–Whitney test answers a different question: it tells us whether the central locations of the two populations are significantly different or, equivalently, whether their relative frequencies are different.

Suppose we randomly select a group of employees and ask each one to complete the same task under two different circumstances: under favorable conditions and under unfavorable conditions, such as a noisy environment. We measure their completion times under both conditions, so we have two measurements for each employee. We want to know if the two times are significantly different, but we can't assume they are normally distributed.

The observations are paired, so we must set `paired=TRUE`:

```
load(file = "./data/workers.rdata")
wilcox.test(fav, unfav, paired = TRUE)
#>
#> Wilcoxon signed rank test
#>
#> data: fav and unfav
#> V = 10, p-value = 1e-04
#> alternative hypothesis: true location shift is not equal to 0
```

The p -value is essentially zero. Statistically speaking, we reject the assumption that the completion times were equal. Practically speaking, it's reasonable to conclude that the times were different.

In this example, setting `paired=TRUE` is critical. Treating the data as unpaired would be wrong because the observations are not independent, and this in turn would produce bogus results. Running the example with `paired=FALSE` produces a p -value of 0.1022, which leads to the wrong conclusion.

See Also

See [Recipe 9.15](#) for the parametric test.

9.17 Testing a Correlation for Significance

Problem

You calculated the correlation between two variables, but you don't know if the correlation is statistically significant.

Solution

The `cor.test` function can calculate both the p -value and the confidence interval of the correlation. If the variables came from normally distributed populations then use the default measure of correlation, which is the Pearson method:

```
cor.test(x, y)
```

For nonnormal populations, use the Spearman method instead:

```
cor.test(x, y, method = "spearman")
```

The function returns several values, including the p -value from the test of significance. Conventionally, $p < 0.05$ indicates that the correlation is likely significant, whereas $p > 0.05$ indicates it is not.

Discussion

In our experience, people often fail to check a correlation for significance. In fact, many people are unaware that a correlation can be insignificant. They jam their data into a computer, calculate the correlation, and blindly believe the result. However, they should ask themselves: Was there enough data? Is the magnitude of the correlation large enough? Fortunately, the `cor.test` function answers those questions.

Suppose we have two vectors, `x` and `y`, with values from normal populations. We might be very pleased that their correlation is greater than 0.75:

```
cor(x, y)
#> [1] 0.751
```

But that is naïve. If we run `cor.test`, it reports a relatively large p -value of 0.09:

```
cor.test(x, y)
#>
#> Pearson's product-moment correlation
#>
#> data: x and y
#> t = 2, df = 4, p-value = 0.09
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#> -0.155 0.971
#> sample estimates:
#> cor
#> 0.751
```

The p -value is above the conventional threshold of 0.05, so we conclude that the correlation is unlikely to be significant.

You can also check the correlation by using the confidence interval. In this example, the confidence interval is $(-0.155, 0.971)$. The interval contains zero and so it is possible that the correlation is zero, in which case there would be no correlation. Again, you could not be confident that the reported correlation is significant.

The `cor.test` output also includes the point estimate reported by `cor` (at the bottom, labeled “sample estimates”), saving you the additional step of running `cor`.

By default, `cor.test` calculates the Pearson correlation, which assumes that the underlying populations are normally distributed. The Spearman method makes no such assumption because it is nonparametric. Use `method="Spearman"` when working with nonnormal data.

See Also

See [Recipe 2.6](#) for calculating simple correlations.

9.18 Testing Groups for Equal Proportions

Problem

You have samples from two or more groups. The groups' elements are binary-valued: either success or failure. You want to know if the groups have equal proportions of successes.

Solution

Use the `prop.test` function with two vector arguments:

```
ns <- c(48, 64)
nt <- c(100, 100)
prop.test(ns, nt)
#>
#> 2-sample test for equality of proportions with continuity
#> correction
#>
#> data: ns out of nt
#> X-squared = 5, df = 1, p-value = 0.03
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#> -0.3058 -0.0142
#> sample estimates:
#> prop 1 prop 2
#> 0.48 0.64
```

These are parallel vectors. The first vector, `ns`, gives the number of successes in each group. The second vector, `nt`, gives the size of the corresponding group (often called the *number of trials*).

The output includes a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that it is likely the groups' proportions are different, whereas a *p*-value exceeding 0.05 provides no such evidence.

Discussion

In [Recipe 9.11](#), we tested a proportion based on one sample. Here, we have samples from multiple groups and want to compare the proportions in the underlying groups.

One of the authors recently taught statistics to 38 students and awarded a grade of A to 14 of them. A colleague taught the same class to 40 students and awarded an A to only 10. We wanted to know: was the author fostering grade inflation by awarding significantly more A grades than the other teacher did?

We used `prop.test`. “Success” means awarding an A, so the vector of successes contains two elements, the number awarded by the author and the number awarded by the colleague:

```
successes <- c(14, 10)
```

The number of trials is the number of students in the corresponding class:

```
trials <- c(38, 40)
```

The `prop.test` output yields a p -value of 0.4:

```
prop.test(successes, trials)
#>
#> 2-sample test for equality of proportions with continuity
#> correction
#>
#> data: successes out of trials
#> X-squared = 0.8, df = 1, p-value = 0.4
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#> -0.111 0.348
#> sample estimates:
#> prop 1 prop 2
#> 0.368 0.250
```

The relatively large p -value means that we cannot reject the null hypothesis: the evidence does not suggest any difference between the teachers’ grading.

See Also

See [Recipe 9.11](#).

9.19 Performing Pairwise Comparisons Between Group Means

Problem

You have several samples, and you want to perform a pairwise comparison between the sample means. That is, you want to compare the mean of every sample against the mean of every other sample.

Solution

Place all data into one vector and create a parallel factor to identify the groups. Use `pairwise.t.test` to perform the pairwise comparison of means:

```
pairwise.t.test(x, f) # x is the data, f is the grouping factor
```

The output contains a table of p -values, one for each pair of groups. Conventionally, if $p < 0.05$ then the two groups likely have different means, whereas $p > 0.05$ provides no such evidence.

Discussion

This is more complicated than [Recipe 9.15](#), where we compared the means of two samples. Here we have several samples and want to compare the mean of every sample against the mean of every other sample.

Statistically speaking, pairwise comparisons are tricky. It is not the same as simply performing a t -test on every possible pair. The p -values must be adjusted, as otherwise you will get an overly optimistic result. The help pages for `pairwise.t.test` and `p.adjust` describe the adjustment algorithms available in R. Anyone doing serious pairwise comparisons is urged to review the help pages and consult a good textbook on the subject.

Suppose we are using a larger sample of the data from [Recipe 5.5](#), where we combined data for freshmen, sophomores, and juniors into a data frame called `comb`. The data frame has two columns: the data in a column called `values`, and the grouping factor in a column called `ind`. We can use `pairwise.t.test` to perform pairwise comparisons between the groups:

```
pairwise.t.test(comb$values, comb$ind)
#>
#>  Pairwise comparisons using t-tests with pooled SD
#>
#>  data: comb$values and comb$ind
#>
#>    fresh soph
#> soph 0.001 -
#> jrs   3e-04 0.592
#>
#> P value adjustment method: holm
```

Notice the table of p -values. The comparisons of juniors versus freshmen and of sophomores versus freshmen produced small p -values: 0.001 and 0.0003, respectively. We can conclude there are significant differences between those groups. However, the comparison of sophomores versus juniors produced a (relatively) large p -value of 0.592, so they are not significantly different.

See Also

See [Recipe 5.5](#) and [Recipe 9.15](#).

9.20 Testing Two Samples for the Same Distribution

Problem

You have two samples, and you are wondering: did they come from the same distribution?

Solution

The Kolmogorov–Smirnov test compares two samples and tests them for being drawn from the same distribution. The `ks.test` function implements that test:

```
ks.test(x, y)
```

The output includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the two samples (x and y) were drawn from different distributions, whereas a p -value exceeding 0.05 provides no such evidence.

Discussion

The Kolmogorov–Smirnov test is wonderful for two reasons. First, it is a nonparametric test and so you needn't make any assumptions regarding the underlying distributions: it works for all distributions. Second, it checks the location, dispersion, and shape of the populations, based on the samples. If these characteristics disagree then the test will detect that, allowing you to conclude that the underlying distributions are different.

Suppose we suspect that the vectors x and y come from differing distributions. Here, `ks.test` reports a p -value of 0.04:

```
ks.test(x, y)
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#>  data: x and y
#> D = 0.2, p-value = 0.04
#> alternative hypothesis: two-sided
```

From the small p -value we can conclude that the samples are from different distributions. However, when we test x against another sample, z , the p -value is much larger (0.6); this suggests that x and z could have the same underlying distribution:

```
z <- rnorm(100, mean = 4, sd = 6)
ks.test(x, z)
#>
#>  Two-sample Kolmogorov-Smirnov test
#>
#>  data: x and z
```

```
#> D = 0.1, p-value = 0.6  
#> alternative hypothesis: two-sided
```


CHAPTER 10

Graphics

Graphics is a great strength of R. The `graphics` package is part of the standard distribution and contains many useful functions for creating a variety of graphic displays. The base functionality has been expanded and made easier with `ggplot2`, part of the `tidyverse` of packages. In this chapter we will focus on examples using `ggplot2`, and we will occasionally suggest other packages. In this chapter's See Also sections we mention functions in other packages that do the same job in a different way. We suggest that you explore those alternatives if you are dissatisfied with what's offered by `ggplot2` or base graphics.

Graphics is a vast subject, and we can only scratch the surface here. Winston Chang's *R Graphics Cookbook, 2nd ed.*, is part of the O'Reilly Cookbook series and walks through many useful recipes with a focus on `ggplot2`. If you want to delve deeper, we recommend *R Graphics* by Paul Murrell (Chapman & Hall); it discusses the paradigms behind R graphics, explains how to use the graphics functions, and contains numerous examples, including the code to re-create them. Some of the examples are pretty amazing.

The Illustrations

The graphs in this chapter are mostly plain and unadorned. We did that intentionally. When you call the `ggplot` function, as in:

```
library(tidyverse)  
df <- data.frame(x = 1:5, y = 1:5)  
ggplot(df, aes(x, y)) +  
  geom_point()
```

you get a plain graphical representation of `x` and `y` as shown in Figure 10-1.

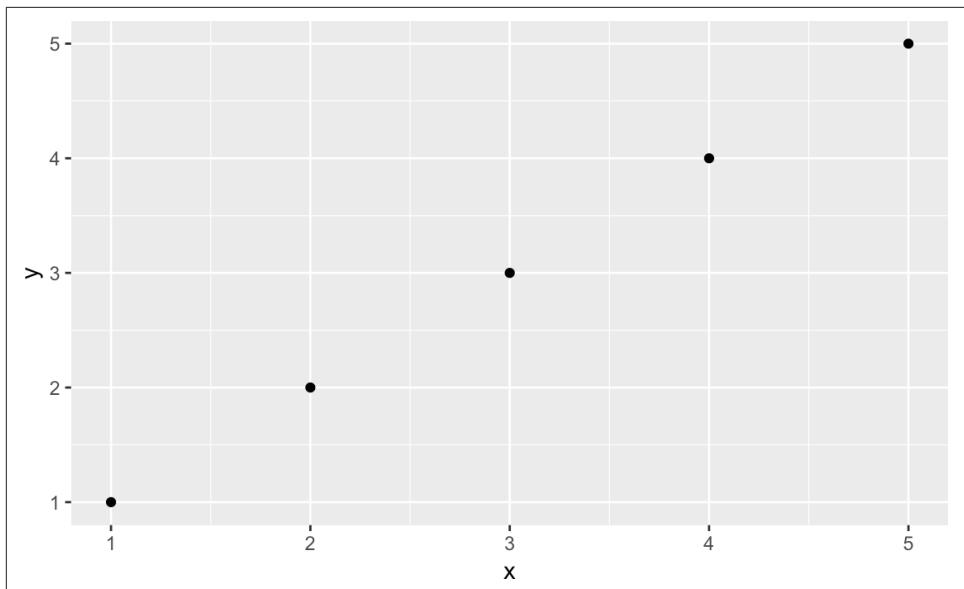


Figure 10-1. Simple plot

You could adorn the graph with colors, a title, labels, a legend, text, and so forth, but then the call to `ggplot` becomes more and more crowded, obscuring the basic intention:

```
ggplot(df, aes(x, y)) +  
  geom_point() +  
  labs(  
    title = "Simple Plot Example",  
    subtitle = "with a subtitle",  
    x = "x-values",  
    y = "y-values"  
) +  
  theme(panel.background = element_rect(fill = "white", color = "grey50"))
```

The resulting plot is shown in Figure 10-2. We want to keep the recipes clean, so we emphasize the basic plot and then show later (as in Recipe 10.2) how to add adornments.

Simple Plot Example

with a subtitle

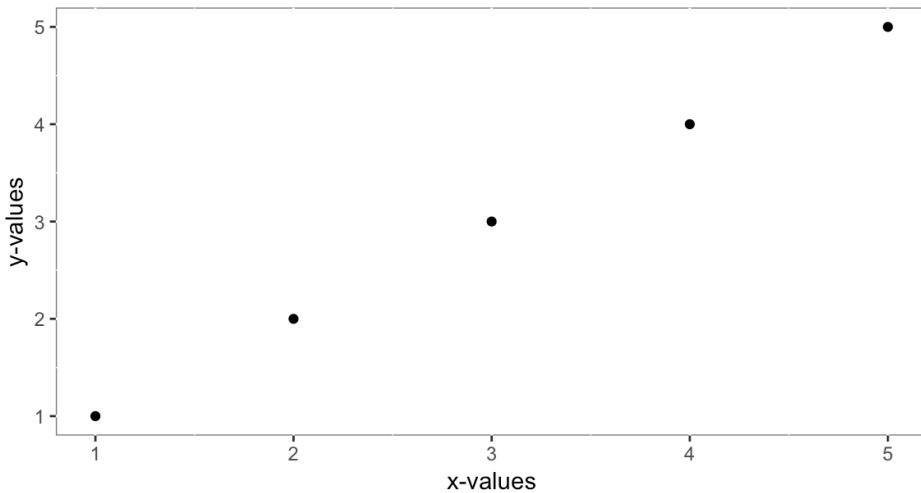


Figure 10-2. Slightly more complicated plot

Notes on ggplot2 Basics

While the package is called `ggplot2`, the primary plotting function in the package is called `ggplot`. It is important to understand the basic pieces of a `ggplot` graph. In the preceding examples, you can see that we pass data into `ggplot`, then define how the graph is created by stacking together small phrases that describe some aspect of the plot. This stacking together of phrases is part of the “grammar of graphics” ethos (that’s where the `gg` comes from). To learn more, you can read [“A Layered Grammar of Graphics”](#) written by `ggplot` author Hadley Wickham. The concept originated with Leland Wilkinson, who articulated the idea of building graphics up from a set of primitives (i.e., verbs and nouns). With `ggplot`, the underlying data need not be fundamentally reshaped for each type of graphical representation. In general, the data stays the same and the user changes the syntax slightly to illustrate the data differently. This is significantly more consistent than base graphics, which often require reshaping the data in order to change the way it is visualized.

As we’re talking about `ggplot` graphics, it’s worth defining the components of a `ggplot` graph:

Geometric object functions

These are geometric objects that describe the type of graph being created. Their names start with `geom_`; examples include `geom_line`, `geom_boxplot`, and `geom_point`, along with dozens more.

Aesthetics

The aesthetics, or aesthetic mappings, communicate to `ggplot` which fields in the source data get mapped to which visual elements in the graphic. This is the `aes` line in a `ggplot` call.

Stats

Stats are statistical transformations that are done before displaying the data. Not all graphs will have stats, but a few common stats are `stat_ecdf` (the empirical cumulative distribution function) and `stat_identity`, which tells `ggplot` to pass the data without doing any stats at all.

Facet functions

Facets are subplots where each small plot represents a subgroup of the data. The faceting functions include `facet_wrap` and `facet_grid`.

Themes

Themes are the visual elements of the plot that are not tied to data. These might include titles, margins, table of contents locations, or font choices.

Layer

A layer is a combination of data, aesthetics, a geometric object, a stat, and other options to produce a visual layer in the `ggplot` graphic.

“Long” Versus “Wide” Data with `ggplot`

One of the first sources of confusion for new `ggplot` users is that they are inclined to reshape their data to be “wide” before plotting it. “Wide” here means every variable they are plotting is its own column in the underlying data frame. This is an approach that many users develop while using Excel and then bring with them to R. `ggplot` works most easily with “long” data, where additional variables are added as rows in the data frame rather than columns. The great side effect of adding more measurements as rows is that any properly constructed `ggplot` graphs will automatically update to reflect the new data without changing the `ggplot` code. If each additional variable were added as a column, then the plotting code would have to be changed to introduce additional variables. This idea of “long” versus “wide” data will become more obvious in the examples in the rest of this chapter.

Graphics in Other Packages

R is highly programmable, and many people have extended its graphics machinery with additional features. Quite often, packages include specialized functions for plotting their results and objects. The `zoo` package, for example, implements a time series object. If you create a `zoo` object `z` and call `plot(z)`, then the `zoo` package does the

plotting; it creates a graphic that is customized for displaying a time series. `zoo` uses base graphics, so the resulting graph will not be a `ggplot` graphic.

There are even entire packages devoted to extending R with new graphics paradigms. The `lattice` package is an alternative to base graphics that predates `ggplot2`. It uses a powerful graphics paradigm that enables you to create informative graphics more easily. It was implemented by Deepayan Sarkar, who also wrote *Lattice: Multivariate Data Visualization with R* (Springer), which explains the package and how to use it. The `lattice` package is also described in *R in a Nutshell* (O'Reilly).

There are two chapters in Hadley Wickham and Garrett Grolemund's excellent book *R for Data Science* that deal with graphics. Chapter 7, "Exploratory Data Analysis," focuses on exploring data with `ggplot2`, while Chapter 28, "Graphics for Communication," explores communicating to others with graphics. *R for Data Science* is available in print or [online](#).

10.1 Creating a Scatter Plot

Problem

You have paired observations: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. You want to create a scatter plot of the pairs.

Solution

We can plot the data by calling `ggplot`, passing in the data frame, and invoking a geometric point function:

```
ggplot(df, aes(x, y)) +  
  geom_point()
```

In this example, the data frame is called `df` and the `x` and `y` data are in fields named `x` and `y`, which we pass to the aesthetic in the call `aes(x, y)`.

Discussion

A scatter plot is a common first attack on a new dataset. It's a quick way to see the relationship, if any, between `x` and `y`.

Plotting with `ggplot` requires telling `ggplot` what data frame to use, then what type of graph to create and which aesthetic mapping (`aes`) to use. The `aes` in this case defines which field from `df` goes into which axis on the plot. Then the command `geom_point` communicates that you want a point graph, as opposed to a line or other type of graphic.

We can use the built-in `mtcars` dataset to illustrate plotting horsepower (`hp`) on the x-axis and fuel economy (`mpg`) on the y-axis:

```
ggplot(mtcars, aes(hp, mpg)) +  
  geom_point()
```

The resulting plot is shown in [Figure 10-3](#).

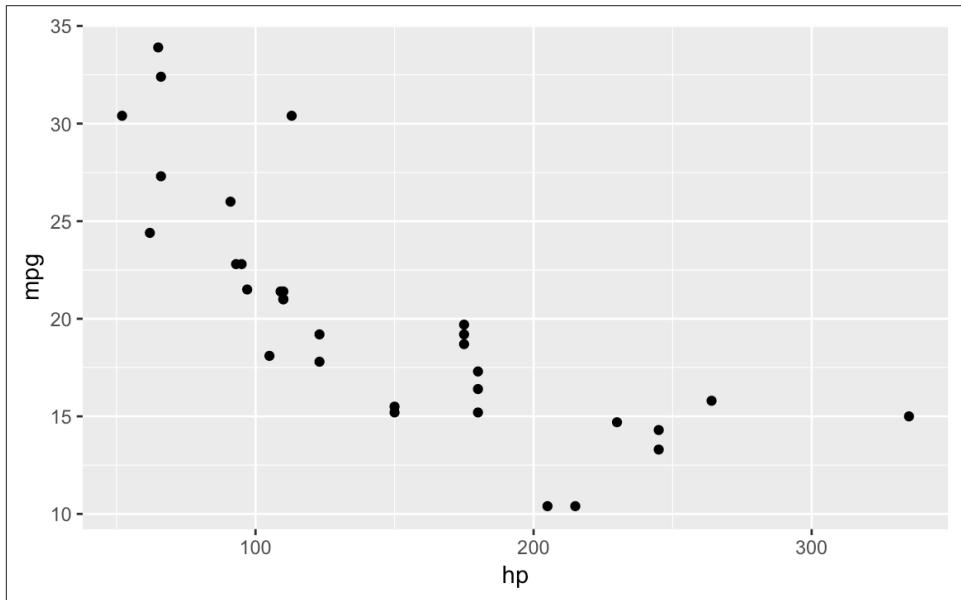


Figure 10-3. Scatter plot

See Also

See [Recipe 10.2](#) for adding a title and labels, [Recipe 10.3](#) for adding a grid, and [Recipe 10.6](#) for adding a legend. See [Recipe 10.8](#) for plotting multiple variables.

10.2 Adding a Title and Labels

Problem

You want to add a title to your plot or add labels for the axes.

Solution

With `ggplot` we add a `labs` element that controls the labels for the title and axes.

When calling `labs` in `ggplot`, specify:

```
title  
Desired title text
```

```
x  
x-axis label
```

```
y  
y-axis label
```

For example:

```
ggplot(df, aes(x, y)) +  
  geom_point() +  
  labs(title = "The Title",  
       x = "X-axis Label",  
       y = "Y-axis Label")
```

Discussion

The graph created in [Recipe 10.1](#) is quite plain. A title and better labels will make it more interesting and easier to interpret.

Note that in `ggplot` you build up the elements of the graph by connecting the parts with the plus sign, `+`. So, we add further graphical elements by stringing together phrases. You can see this in the following code, which uses the built-in `mtcars` dataset and plots horsepower versus fuel economy in a scatter plot, shown in [Figure 10-4](#):

```
ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() +  
  labs(title = "Cars: Horsepower vs. Fuel Economy",  
       x = "HP",  
       y = "Economy (miles per gallon)")
```

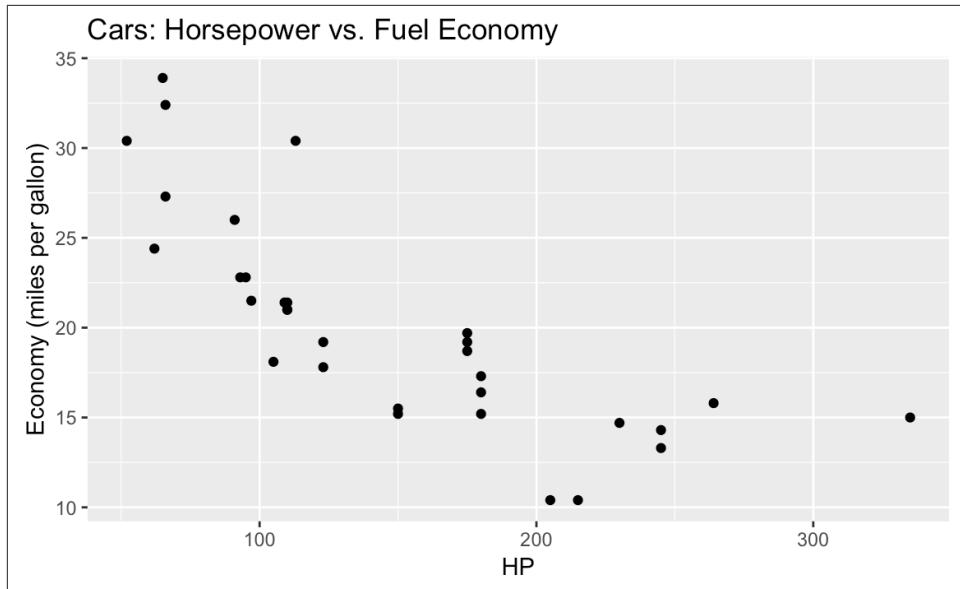


Figure 10-4. Labeled axes and title

10.3 Adding (or Removing) a Grid

Problem

You want to change the background grid of your graphic.

Solution

With `ggplot` background grids come as a default, as you have seen in previous recipes. However, we can alter the background grid using the `theme` function or by applying a prepackaged theme to our graph.

We can use `theme` to alter the background panel of our graphic. This example removes it, as seen in [Figure 10-5](#):

```
ggplot(df) +  
  geom_point(aes(x, y)) +  
  theme(panel.background = element_rect(fill = "white", color = "grey50"))
```

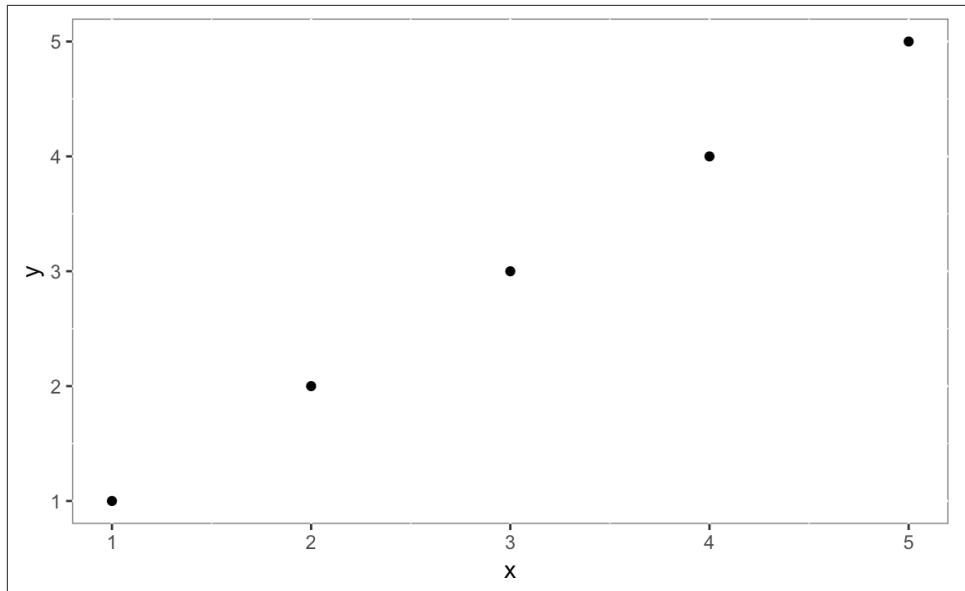


Figure 10-5. White background

Discussion

`ggplot` fills in the background with a grey grid by default. You may find yourself wanting to remove that grid completely or change it to something else. Let's create a `ggplot` graphic and then incrementally change the background style.

We can add or change aspects of our graphic by creating a `ggplot` object, then calling the object and using the `+` to add to it. The background shading in a `ggplot` graphic is actually three different graph elements:

`panel.grid.major`

The major grid is white by default and heavy.

`panel.grid.minor`

The minor grid is white by default and light.

`panel.background`

The background is grey by default.

You can see these elements if you look carefully at the background of [Figure 10-4](#).

If we set the background as `element_blank`, then the major and minor grids are still there, but they are white on white so we can't see them in [Figure 10-6](#):

```
g1 <- ggplot(mtcars, aes(hp, mpg)) +  
  geom_point() +  
  labs(title = "Cars: Horsepower vs. Fuel Economy",  
       x = "HP",  
       y = "Economy (miles per gallon)") +  
  theme(panel.background = element_blank())  
g1
```

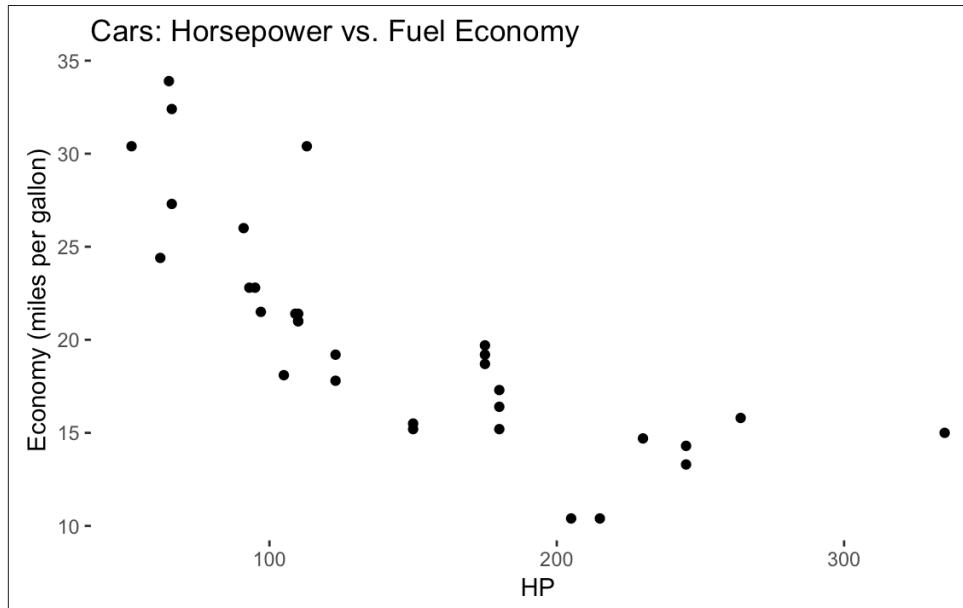


Figure 10-6. Blank background

Notice in the previous code we put the `ggplot` graph into a variable called `g1`. Then we printed the graphic by just calling `g1`. Having the graph inside of `g1` means we can add further graphical components without rebuilding the graph.

If we wanted to show the background grid with unusual patterns for illustration, it's as easy as setting its components to a color and setting a line type, as in this example (see [Figure 10-7](#)):

```
g2 <- g1 + theme(panel.grid.major =  
  element_line(color = "black", linetype = 3)) +  
  # linetype = 3 is dash  
  theme(panel.grid.minor =  
  element_line(color = "darkgrey", linetype = 4))  
  # linetype = 4 is dot dash  
g2
```

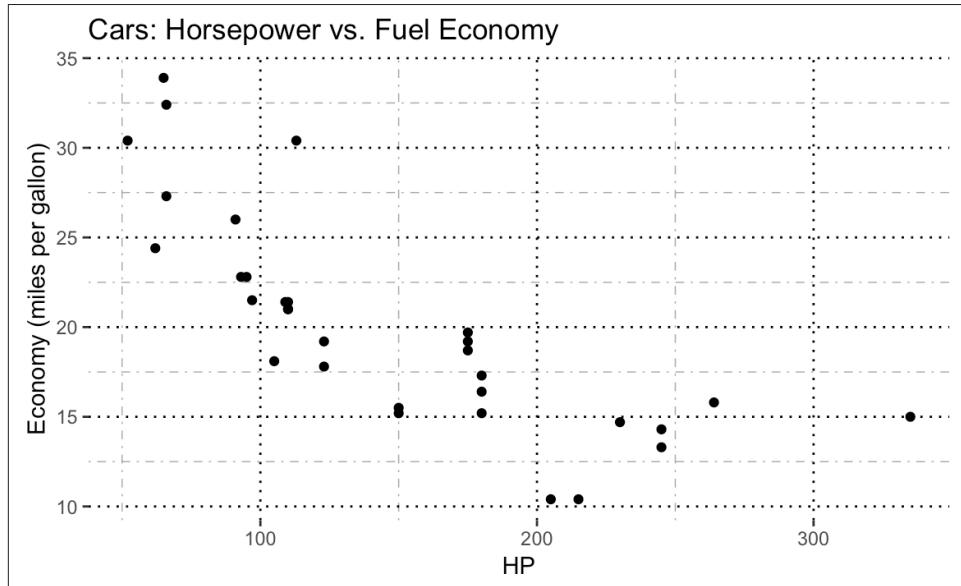


Figure 10-7. Major and minor gridlines

[Figure 10-7](#) lacks visual appeal, but you can clearly see that the dotted black lines make up the major grid and the dashed grey lines are the minor grid.

Or we could do something less garish and take the ggplot object g1 from before and add grey gridlines to the white background, as shown in [Figure 10-8](#):

```
g1 +  
  theme(panel.grid.major = element_line(color = "grey"))
```

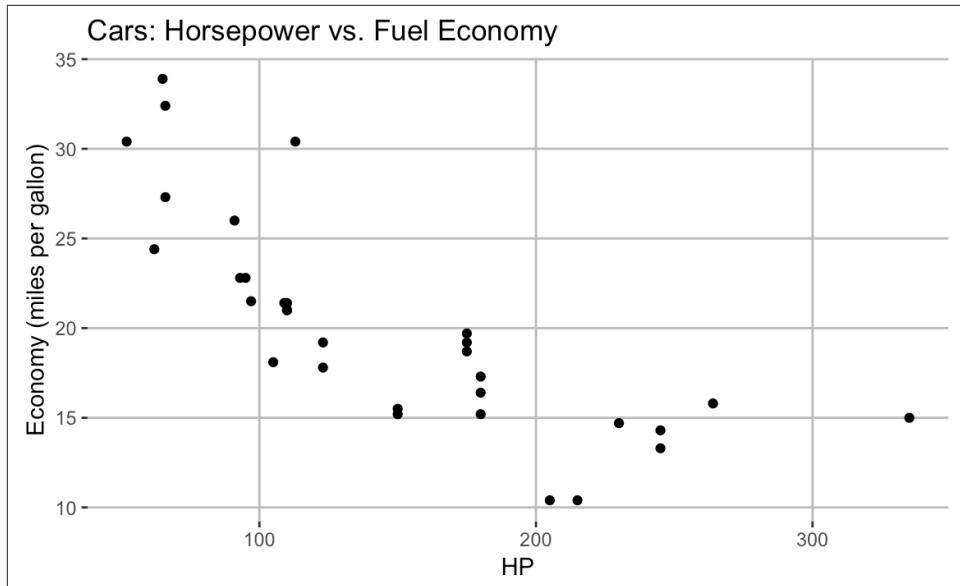


Figure 10-8. Grey major gridlines

See Also

See [Recipe 10.4](#) to see how to apply an entire canned theme to your figure.

10.4 Applying a Theme to a ggplot Figure

Problem

You want your plot to use a preset collection of colors, styles, and formatting.

Solution

ggplot supports *themes*, which are collections of settings for your figures. To use one of the themes, just add the desired theme function to your ggplot with a +:

```
ggplot(df, aes(x, y)) +  
  geom_point() +  
  theme_bw()
```

The ggplot2 package contains the following themes:

```
theme_bw()  
theme_dark()  
theme_classic()  
theme_gray()  
theme_linedraw()
```

```
theme_light()  
theme_minimal()  
theme_test()  
theme_void()
```

Discussion

Let's start with a simple plot and then show how it looks with a few of the built-in themes. [Figure 10-9](#) shows a basic ggplot figure with no theme applied:

```
p <- ggplot(mtcars, aes(x = disp, y = hp)) +  
  geom_point() +  
  labs(title = "mtcars: Displacement vs. Horsepower",  
       x = "Displacement (cubic inches)",  
       y = "Horsepower")  
p
```

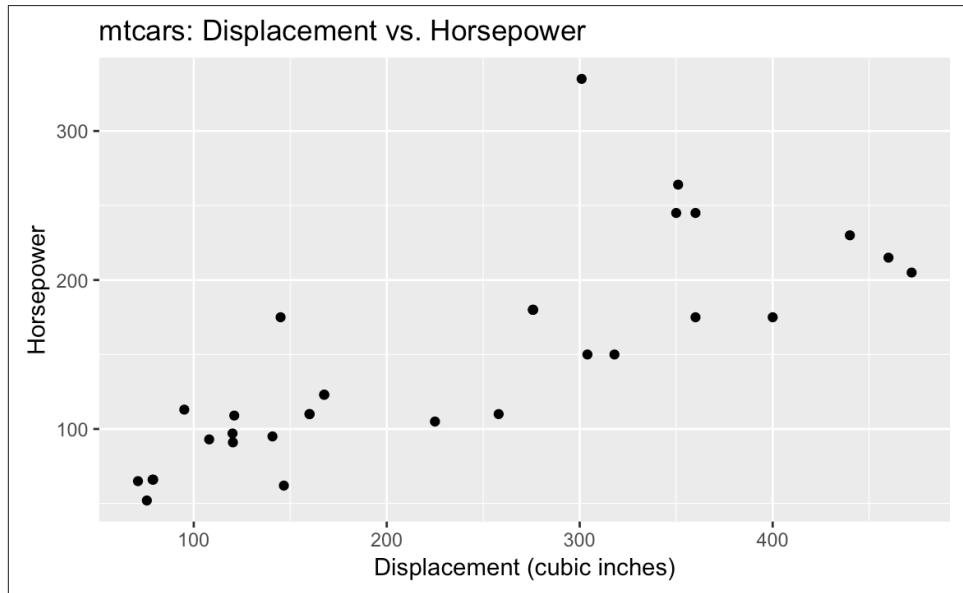


Figure 10-9. Starting plot

Let's create the same plot multiple times, but apply a different theme to each one. [Figure 10-10](#) shows what it looks like with the black and white theme applied:

```
p + theme_bw()
```

[Figure 10-11](#) shows the classic theme:

```
p + theme_classic()
```

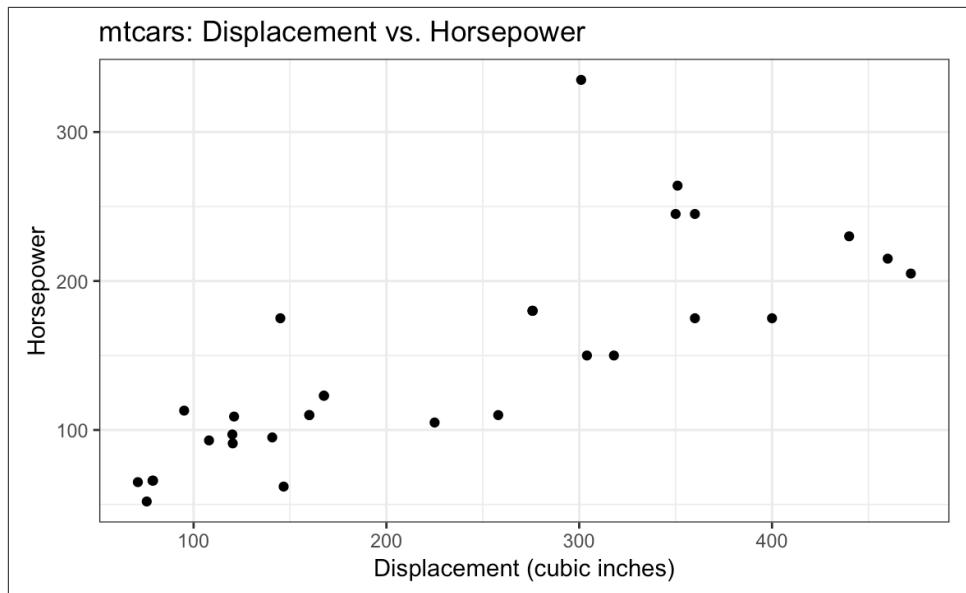


Figure 10-10. *theme_bw*

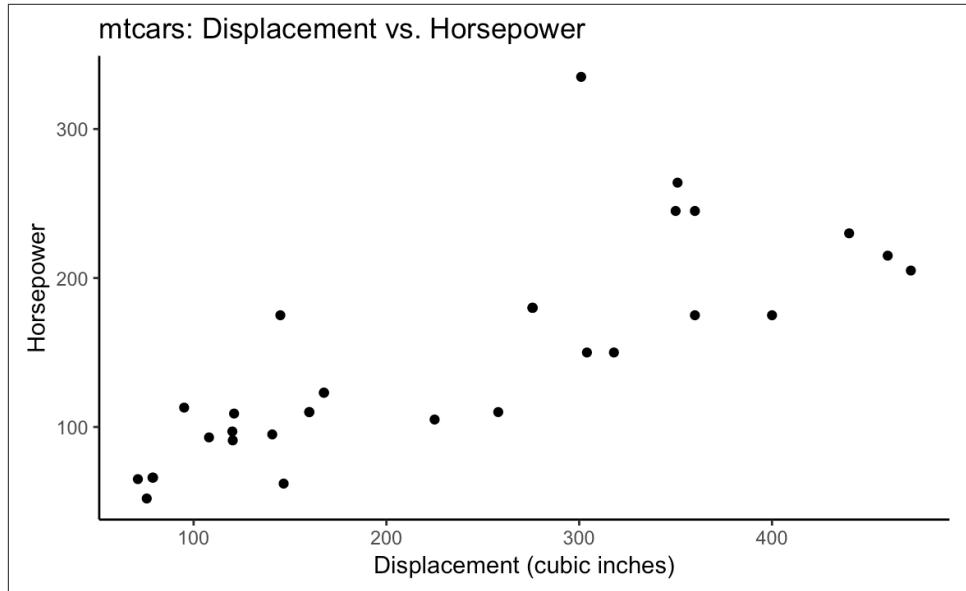


Figure 10-11. *theme_classic*

Figure 10-12 shows the minimal theme:

```
p + theme_minimal()
```

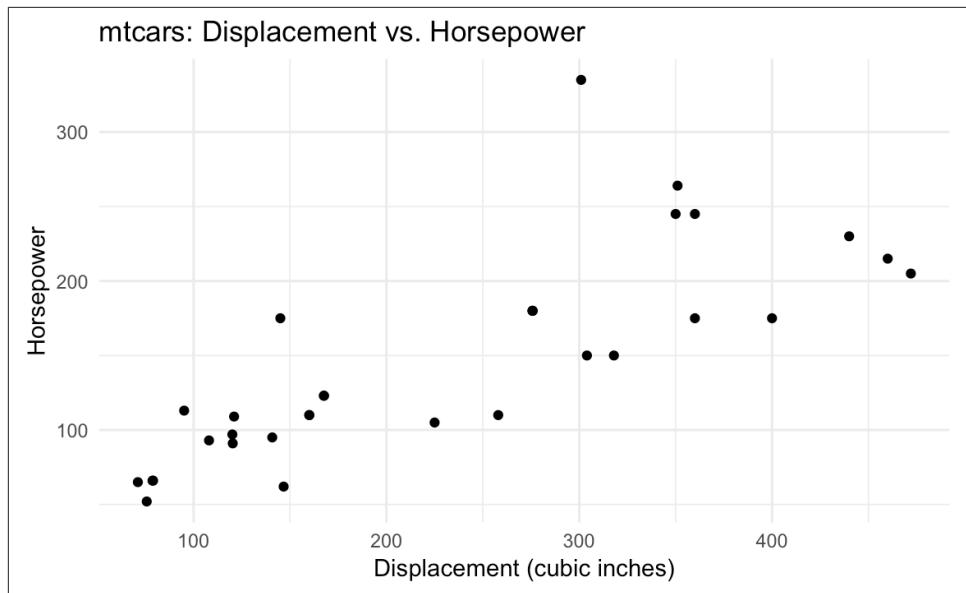


Figure 10-12. `theme_minimal`

And Figure 10-13 shows the void theme:

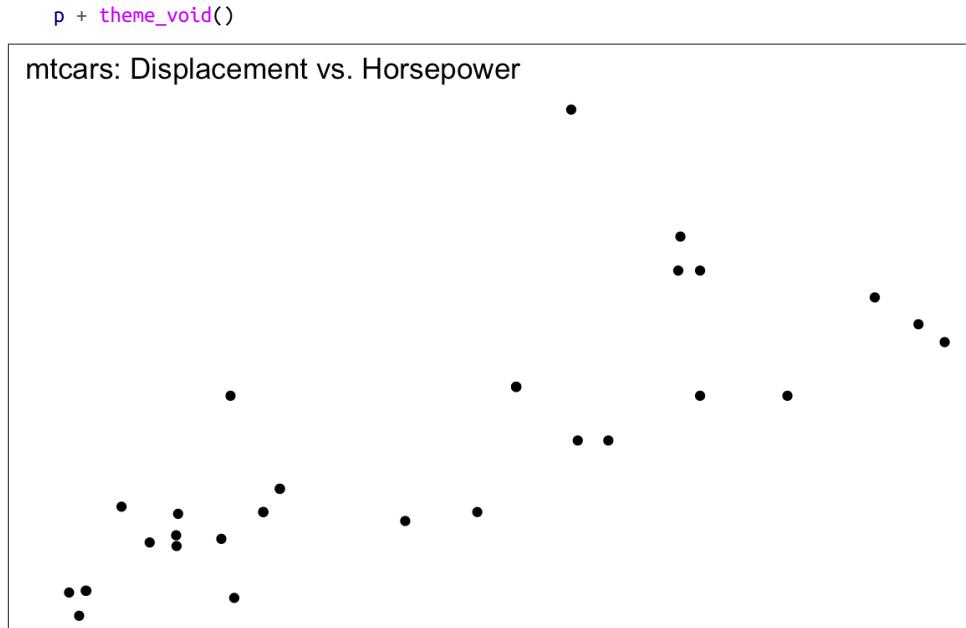


Figure 10-13. `theme_void`

In addition to the themes included in `ggplot2`, there are packages, like `ggtheme`, that include themes to help you make your figures look more like the figures found in popular tools and publications such as Stata or *The Economist*.

See Also

See [Recipe 10.3](#) to see how to change a single theme element.

10.5 Creating a Scatter Plot of Multiple Groups

Problem

You have data in a data frame with multiple observations per record: x , y , and a factor f that indicates the group. You want to create a scatter plot of x and y that distinguishes among the groups.

Solution

With `ggplot` we control the mapping of shapes to the factor f by passing `shape = f` to the `aes` function:

```
ggplot(df, aes(x, y, shape = f)) +  
  geom_point()
```

Discussion

Plotting multiple groups in one scatter plot creates an uninformative mess unless we distinguish one group from another. We make this distinction in `ggplot` by setting the `shape` parameter of the `aes` function.

The built-in `iris` dataset contains paired measures of `Petal.Length` and `Petal.Width`. Each measurement also has a `Species` property indicating the species of the flower that was measured. If we plot all the data at once, we just get the scatter plot shown in [Figure 10-14](#):

```
ggplot(data = iris,  
       aes(x = Petal.Length,  
            y = Petal.Width)) +  
  geom_point()
```

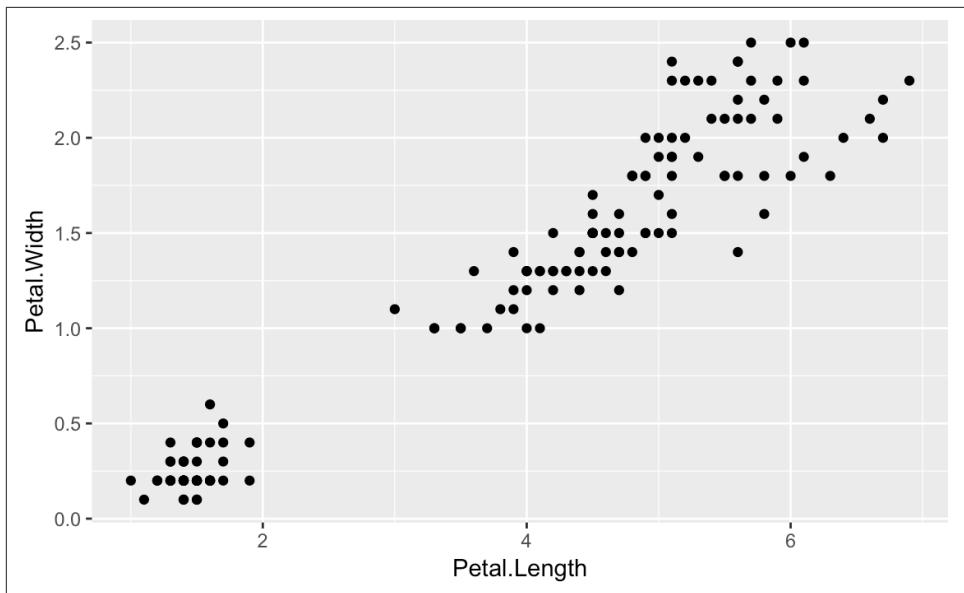


Figure 10-14. *iris*: length vs. width

The graphic would be far more informative if we distinguished the points by species. In addition to distinguishing the species by shape, we could also differentiate by color. We can add `shape = Species` and `color = Species` to our `aes` call to get each species with a different shape and color, as shown in [Figure 10-15](#):

```
ggplot(data = iris,
  aes(
    x = Petal.Length,
    y = Petal.Width,
    shape = Species,
    color = Species
  )) +
  geom_point()
```

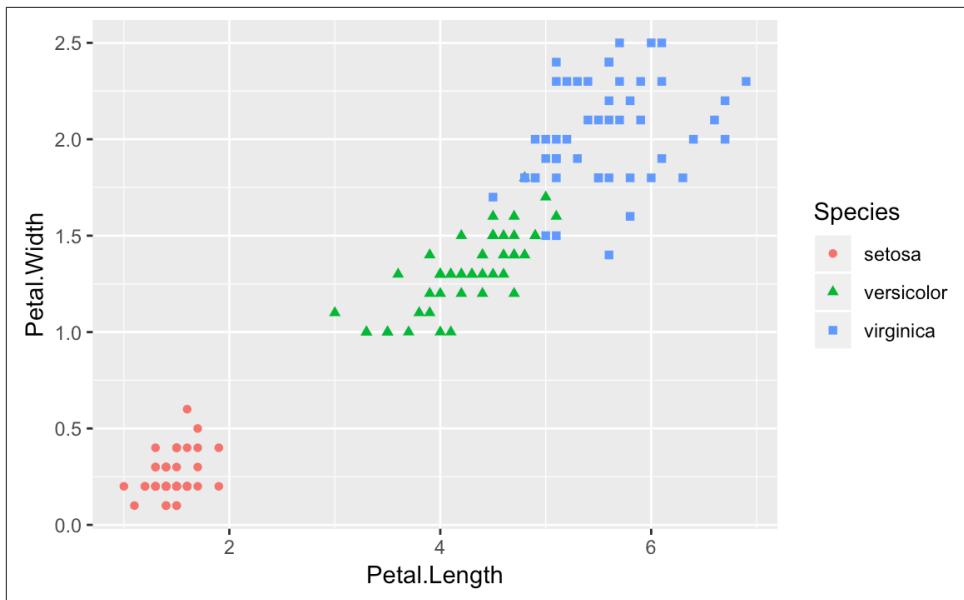


Figure 10-15. *iris*: shape and color

`ggplot` conveniently sets up a legend for you as well, which is handy.

See Also

See [Recipe 10.6](#) for more on how to add a legend.

10.6 Adding (or Removing) a Legend

Problem

You want your plot to include a *legend*, the little box that decodes the graphic for the viewer.

Solution

In most cases `ggplot` will add legends automatically, as you can see in the previous recipe. But if we do not have explicit grouping in the `aes` function, then `ggplot` will not show a legend by default. If we want to force `ggplot` to show a legend, we can set the shape or line type of our graph to a constant. `ggplot` will then show a legend with one group. We use `guides` to guide `ggplot` in how to label the legend.

This can be illustrated with our `iris` scatter plot:

```

g <- ggplot(data = iris,
             aes(x = Petal.Length,
                 y = Petal.Width,
                 shape="Observation")) +
  geom_point() +
  guides(shape=guide_legend(title="My Legend Title"))
g

```

Figure 10-16 illustrates the result of setting the shape to a string value and then relabeling the legend using guides.

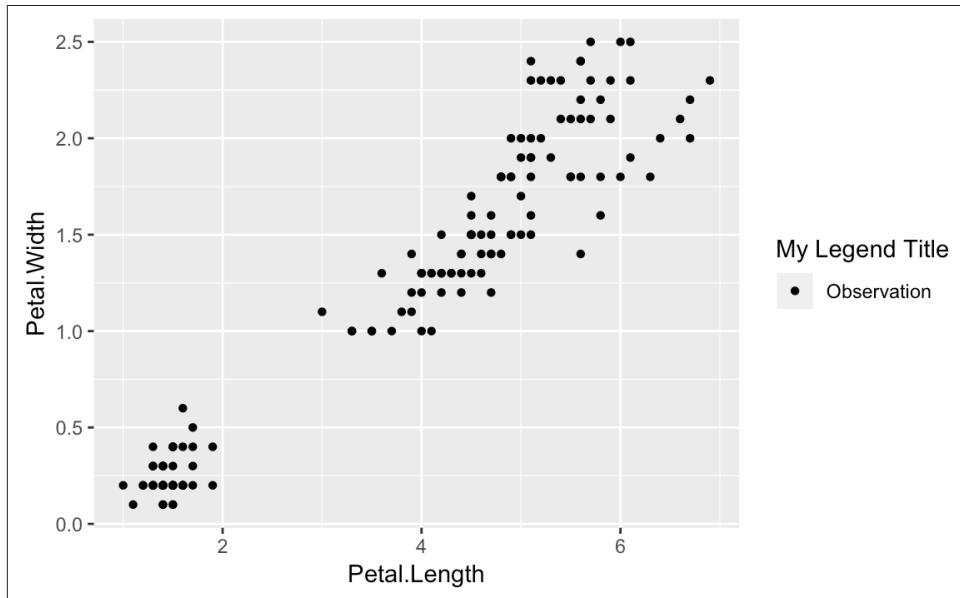


Figure 10-16. Legend added

More commonly, you may want to turn legends off, which you can do by calling `theme` with `legend.position = "none"`. **Figure 10-17** shows the result when we add this call to the `iris` plot from the previous recipe:

```

g <- ggplot(data = iris,
             aes(
               x = Petal.Length,
               y = Petal.Width,
               shape = Species,
               color = Species
             )) +
  geom_point() +
  theme(legend.position = "none")
g

```

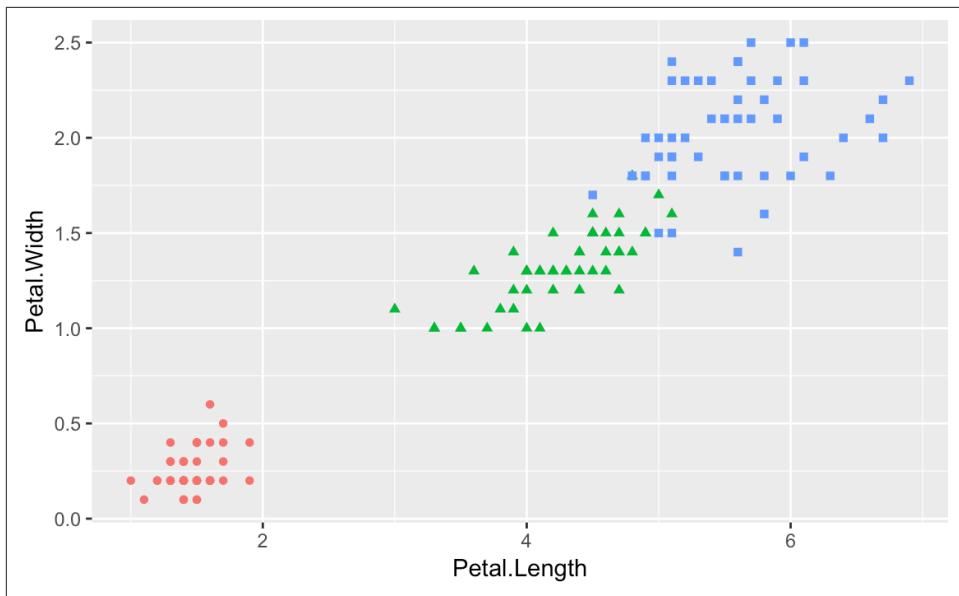


Figure 10-17. Legend removed

Discussion

Adding legends to `ggplot` when there is no grouping is an exercise in “tricking” `ggplot` into showing the legend by passing a string to a grouping parameter in `aes`. While this will not change the grouping (as there is only one group), it will result in a legend being shown with a name.

Then we can use `guides` to alter the legend title. It’s worth noting that we are not changing anything about the data, just exploiting settings in order to coerce `ggplot` into showing a legend when it typically would not.

One of the huge benefits of `ggplot` is its very good defaults. Getting positions and correspondence between labels and their point types is done automatically, but this can be overridden if needed. To remove a legend totally, we set theme parameters with `theme(legend.position = "none")`. We can also set the `legend.position` to be `"left"`, `"right"`, `"bottom"`, `"top"`, or a two-element numeric vector. Use a two-element numeric vector in order to pass `ggplot` specific coordinates of where you want the legend. If you’re using the coordinate positions, the values passed are between 0 and 1 for the `x` and `y` positions, in that order.

Figure 10-18 shows an example of a legend positioned at the bottom, created with this adjustment to the `legend.position`:

```
g + theme(legend.position = "bottom")
```

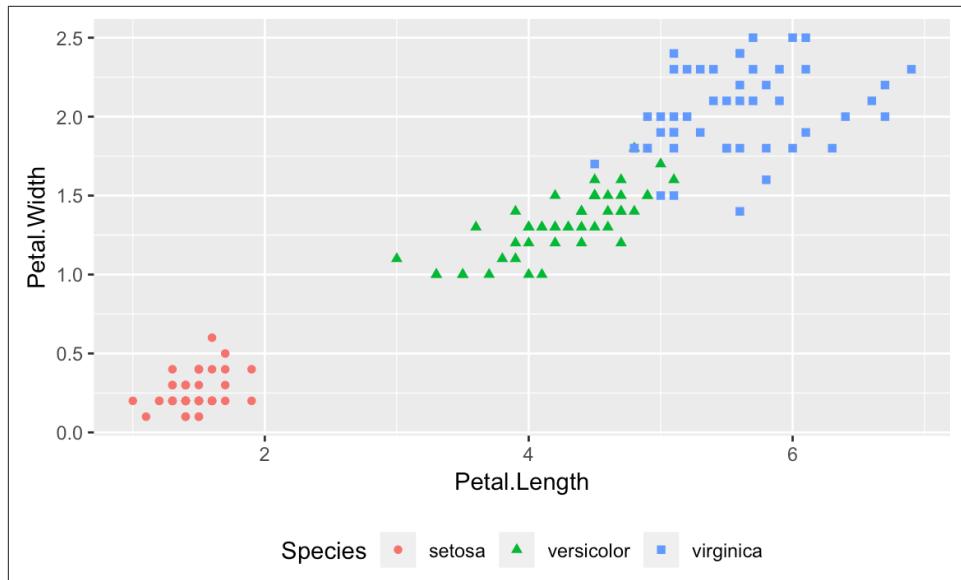


Figure 10-18. Legend at the bottom

Or we could use a two-element numeric vector to put the legend in a specific location, as in **Figure 10-19**. This example puts the center of the legend at 80% to the right and 20% up from the bottom:

```
g + theme(legend.position = c(.8, .2))
```

In many aspects beyond legends, `ggplot` uses sane defaults but offers the flexibility to override them and tweak the details. You can find more details on `ggplot` options related to legends in the help for `theme` by typing `?theme` or by looking in the [ggplot online reference material](#).

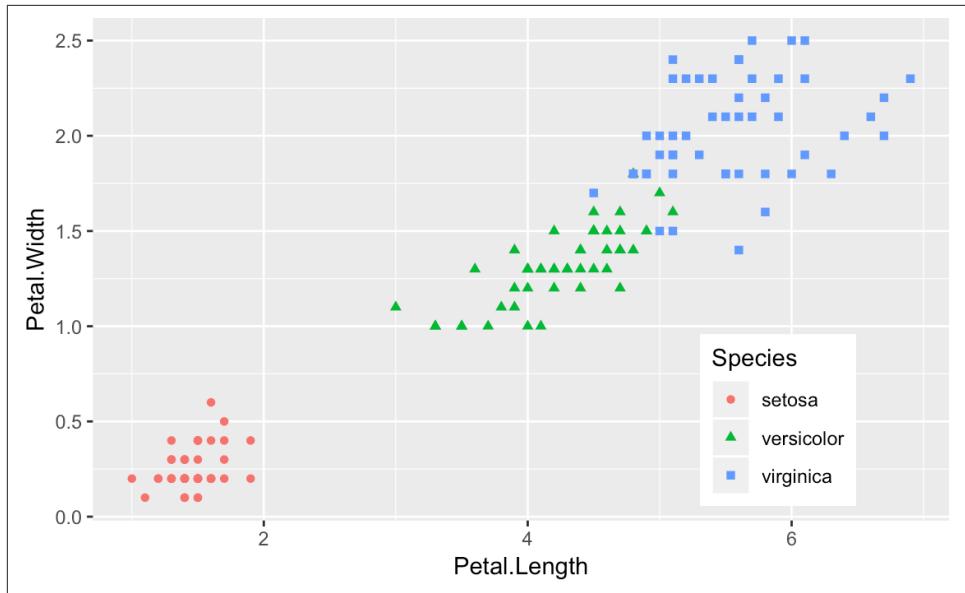


Figure 10-19. Legend at a point

10.7 Plotting the Regression Line of a Scatter Plot

Problem

You are plotting pairs of data points, and you want to add a line that illustrates their linear regression.

Solution

With `ggplot` there is no need to calculate the linear model first using the R `lm` function. We can instead use the `geom_smooth` function to calculate the linear regression inside of our `ggplot` call.

If our data is in a data frame `df` and the `x` and `y` data are in columns `x` and `y`, we plot the regression line like this:

```
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE)
```

The `se = FALSE` parameter tells `ggplot` not to plot the standard error bands around our regression line.

Discussion

Suppose we are modeling the `strongx` dataset found in the `faraway` package. We can create a linear model using the built-in `lm` function in R. We can predict the variable `crossx` as a linear function of `energy`. First, let's look at a simple scatter plot of our data (Figure 10-20):

```
library(faraway)
data(strongx)

ggplot(strongx, aes(energy, crossx)) +
  geom_point()
```

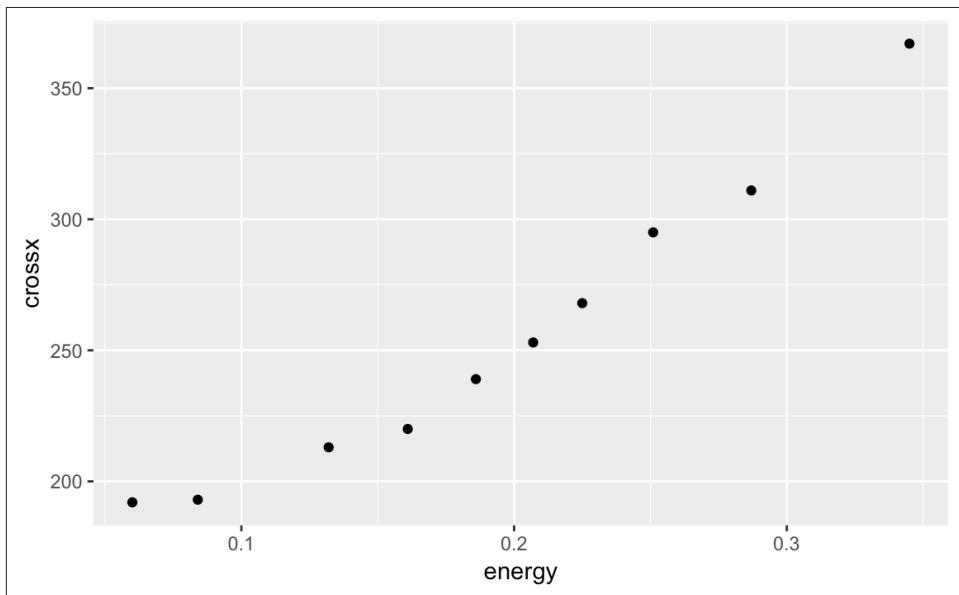


Figure 10-20. *strongx* scatter plot

`ggplot` can calculate a linear model on the fly and then plot the regression line along with our data (Figure 10-21):

```
g <- ggplot(strongx, aes(energy, crossx)) +
  geom_point()

g + geom_smooth(method = "lm",
  formula = y ~ x)
```

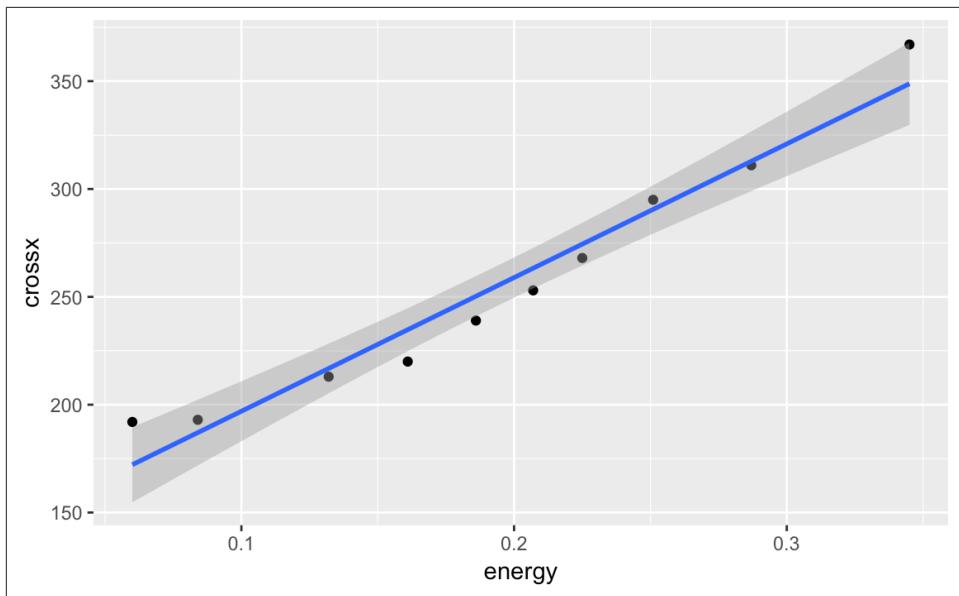


Figure 10-21. Simple linear model ggplot

We can turn the confidence bands off by adding the `se = FALSE` option, as shown in Figure 10-22:

```
g + geom_smooth(method = "lm",
                 formula = y ~ x,
                 se = FALSE)
```

Notice that in `geom_smooth` we use `x` and `y` rather than the variable names. `ggplot` has set `x` and `y` inside the plot based on the aesthetic. Multiple smoothing methods are supported by `geom_smooth`. You can explore those and other options in the help by typing `?geom_smooth`.

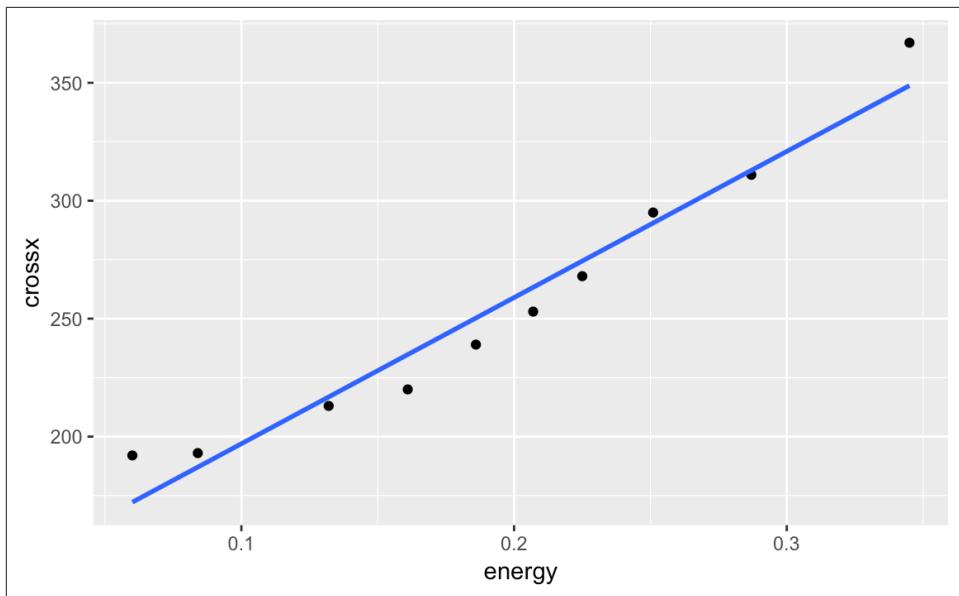


Figure 10-22. Simple linear model ggplot without se

If we had a line we wanted to plot that was stored in another R object, we could use `geom_abline` to plot the line on our graph. In the following example we pull the intercept term and the slope from the regression model `m` and add those to our graph (see Figure 10-23):

```
m <- lm(crossx ~ energy, data = strongx)

ggplot(strongx, aes(energy, crossx)) +
  geom_point() +
  geom_abline(
    intercept = m$coefficients[1],
    slope = m$coefficients[2]
  )
```

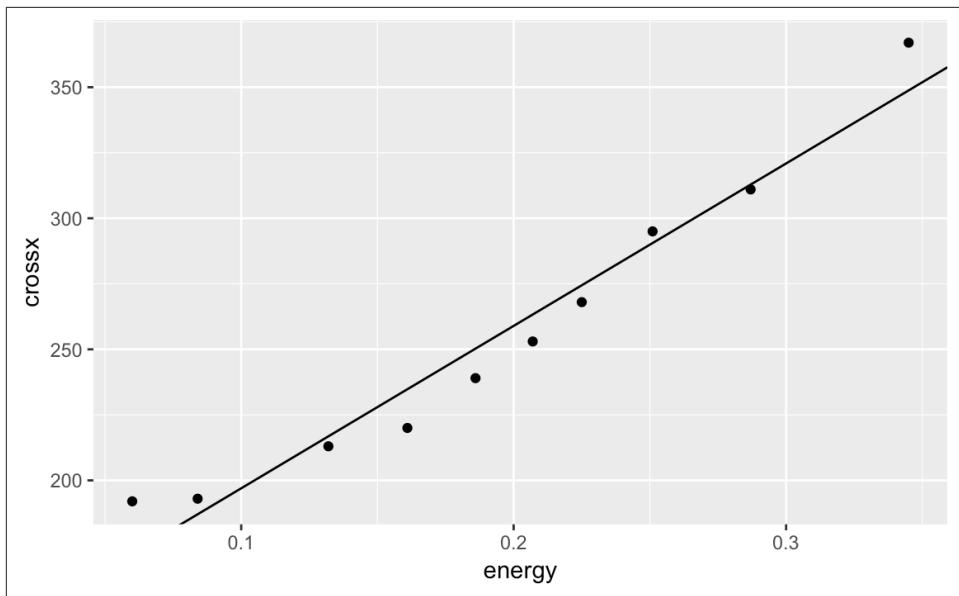


Figure 10-23. Simple line from slope and intercept

This produces a plot very similar to [Figure 10-22](#). The `geom_abline` method can be handy if you are plotting a line from a source other than a simple linear model.

See Also

See [Chapter 11](#) for more about linear regression and the `lm` function.

10.8 Plotting All Variables Against All Other Variables

Problem

Your dataset contains multiple numeric variables. You want to see scatter plots for all pairs of variables.

Solution

`ggplot` does not have any built-in method to create pairs plots; however, the package `GGally` provides this functionality with the `ggpairs` function:

```
library(GGally)
ggpairs(df)
```

Discussion

When you have a large number of variables, finding interrelationships between them is difficult. One useful technique is looking at scatter plots of all pairs of variables. This would be quite tedious if coded pair-by-pair, but the `ggpairs` function from the package `GGally` provides an easy way to produce all those scatter plots at once.

The `iris` dataset contains four numeric variables and one categorical variable:

```
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
```

What is the relationship, if any, between the columns? Plotting the columns with `ggpairs` produces multiple scatter plots, as seen in [Figure 10-24](#):

```
library(GGally)
ggpairs(iris)
```

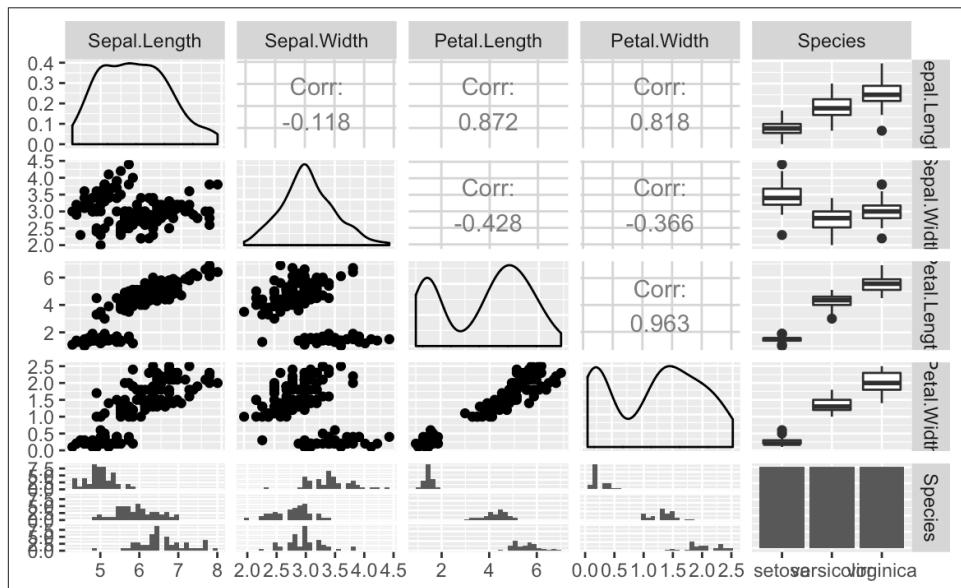


Figure 10-24. ggpairs plot of iris data

The `ggpairs` function is pretty, but not particularly fast. If you're just doing interactive work and want a quick peek at the data, the base R `plot` function provides faster output (see [Figure 10-25](#)):

```
plot(iris)
```

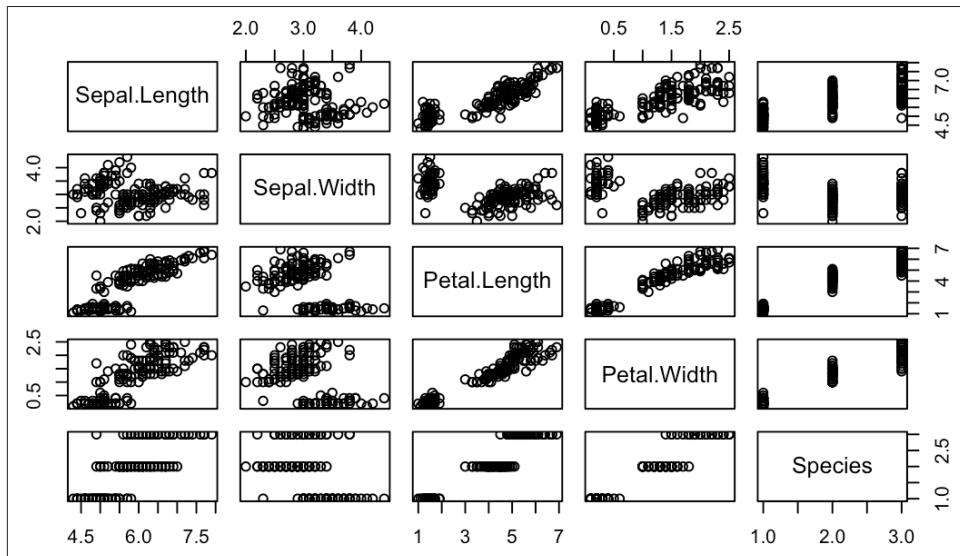


Figure 10-25. Base plot pairs plot

While the `ggpairs` function is not as fast to plot as the Base R `plot` function, it produces density graphs on the diagonal and reports correlation in the upper triangle of the graph. When factors or character columns are present, `ggpairs` produces histograms in the lower triangle of the graph and boxplots in the upper triangle. These are nice additions to understanding relationships in your data.

10.9 Creating One Scatter Plot for Each Group

Problem

Your dataset contains (at least) two numeric variables and a factor or character field defining a group. You want to create several scatter plots for the numeric variables, with one scatter plot for each level of the factor or character field.

Solution

We produce this kind of plot, called a *conditioning plot*, in `ggplot` by adding `facet_wrap` to our plot. In this example we use the data frame `df`, which contains three columns, `x`, `y`, and `f`, with `f` being a factor (or a character string):

```
ggplot(df, aes(x, y)) +  
  geom_point() +  
  facet_wrap(~ f)
```

Discussion

Conditioning plots (coplots) are another way to explore and illustrate the effect of a factor or to compare different groups to each other.

The `Cars93` dataset contains 27 variables describing 93 car models as of 1993. Two numeric variables are `MPG.city`, the miles per gallon in the city, and `Horsepower`, the engine horsepower. One categorical variable is `Origin`, which can be USA or non-USA according to where the model was built.

Exploring the relationship between MPG and horsepower, we might ask: is there a different relationship for USA models and non-USA models?

Let's examine this as a facet plot (Figure 10-26):

```
data(Cars93, package = "MASS")
ggplot(Cars93, aes(MPG.city, Horsepower)) +
  geom_point() +
  facet_wrap(~ Origin)
```

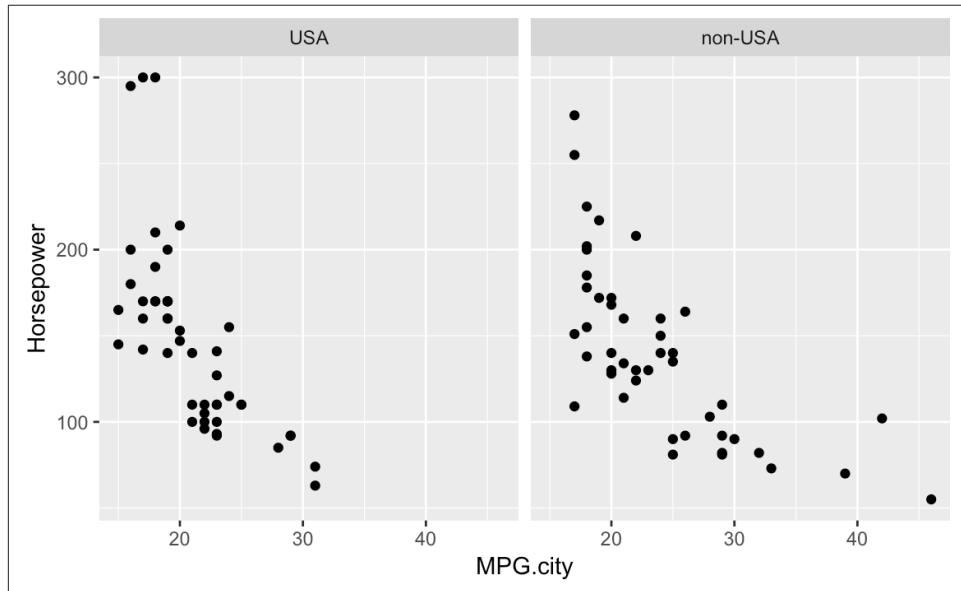


Figure 10-26. Cars93 data with facet

The resulting plot reveals a few insights. If we really crave that 300-horsepower monster, then we'll have to buy a car built in the USA; but if we want high MPG, we have more choices among non-USA models. These insights could be teased out of a statistical analysis, but the visual presentation reveals them much more quickly.

Note that using `facet` results in subplots with the same x- and y-axis ranges. This helps ensure that visual inspection of the data is not misleading because of differing axis ranges.

See Also

The Base R graphics function `coplot` can accomplish very similar plots using only base graphics.

10.10 Creating a Bar Chart

Problem

You want to create a bar chart.

Solution

A common situation is to have a column of data that represents a group and then another column that represents a measure about that group. This format is “long” data because the data runs vertically instead of having a column for each group.

Using the `geom_bar` function in `ggplot`, we can plot the heights as bars. If the data is already aggregated, we add `stat = "identity"` so that `ggplot` knows it needs to do no aggregation on the groups of values before plotting:

```
ggplot(data = df, aes(x, y)) +  
  geom_bar(stat = "identity")
```

Discussion

Let’s use the cars made by Ford in the `Cars93` dataset in an example:

```
ford_cars <- Cars93 %>%  
  filter(Manufacturer == "Ford")  
  
ggplot(ford_cars, aes(Model, Horsepower)) +  
  geom_bar(stat = "identity")
```

Figure 10-27 shows the resulting bar chart.

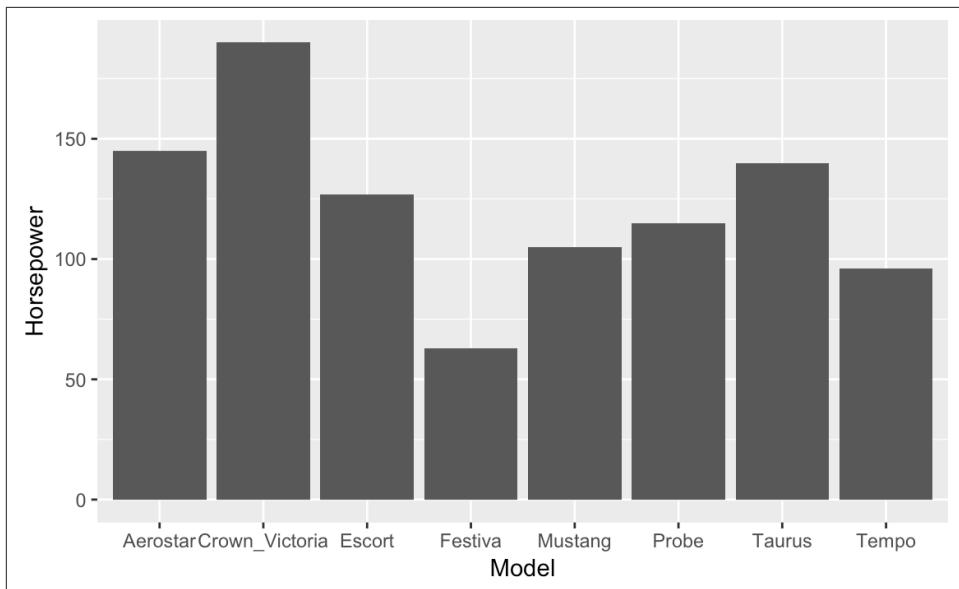


Figure 10-27. Ford cars bar chart

This example uses `stat = "identity"`, which assumes that the heights of your bars are conveniently stored as a value in one field with only one record per column. That is not always the case, however. Often you have a vector of numeric data and a parallel factor or character field that groups the data, and you want to produce a bar chart of the group means or the group totals.

Let's work up an example using the built-in `airquality` dataset, which contains daily temperature data for a single location for five months. The data frame has a numeric `Temp` column and `Month` and `Day` columns. If we want to plot the mean temperature by month using `ggplot`, we don't need to precompute the mean; instead, we can have `ggplot` do that in the plot command logic. To tell `ggplot` to calculate the mean, we pass `stat = "summary"`, `fun.y = "mean"` to the `geom_bar` command. We can also turn the month numbers into dates using the built-in constant `month.abb`, which contains the abbreviations for the months:

```
ggplot(airquality, aes(month.abb[Month], Temp)) +
  geom_bar(stat = "summary", fun.y = "mean") +
  labs(title = "Mean Temp by Month",
       x = "",
       y = "Temp (deg. F)")
```

[Figure 10-28](#) shows the resulting plot. But you might notice the sort order on the months is alphabetical, which is not how we typically like to see months sorted.

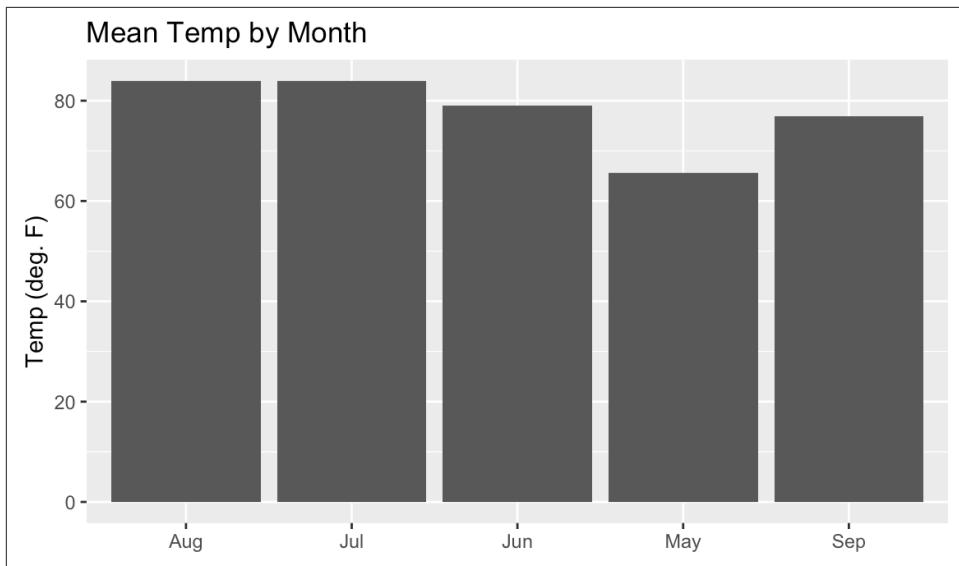


Figure 10-28. Bar chart: temp by month

We can fix the sorting issue using a few functions from `dplyr` combined with `fct_inorder` from the `forcats` tidyverse package. To get the months in the correct order, we can sort the data frame by `Month`, which is the month number. Then we can apply `fct_inorder`, which will arrange our factors in the order they appear in the data. You can see in [Figure 10-29](#) that the bars are now sorted properly:

```
library(forcats)

aq_data <- airquality %>%
  arrange(Month) %>%
  mutate(month_abb = fct_inorder(month.abb[Month]))

ggplot(aq_data, aes(month_abb, Temp)) +
  geom_bar(stat = "summary", fun.y = "mean") +
  labs(title = "Mean Temp by Month",
       x = "",
       y = "Temp (deg. F)")
```

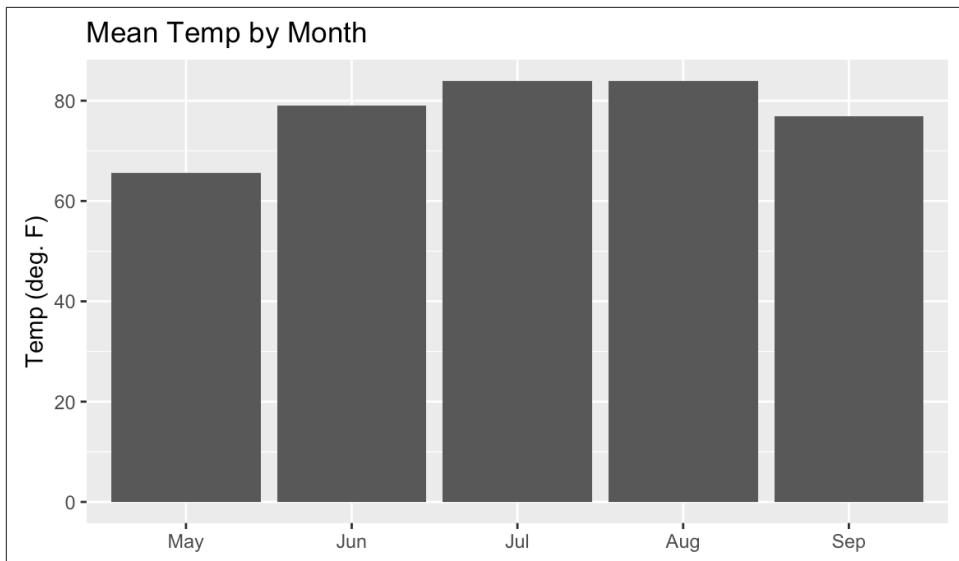


Figure 10-29. Bar chart properly sorted

See Also

See [Recipe 10.11](#) for adding confidence intervals and [Recipe 10.12](#) for adding color.

Type `?geom_bar` for help with bar charts in `ggplot`.

You can also use `barplot` for Base R bar charts or the `barchart` function in the `lattice` package.

10.11 Adding Confidence Intervals to a Bar Chart

Problem

You want to augment a bar chart with confidence intervals.

Solution

Suppose we have a data frame `df` with columns `group` (group names), `stat` (a column of statistics), and `lower` and `upper` (which represent the corresponding limits for the confidence intervals). We can display a bar chart of `stat` for each `group` and its confidence interval using the `geom_bar` function combined with `geom_errorbar`:

```
ggplot(df, aes(group, stat)) +
  geom_bar(stat = "identity") +
  geom_errorbar(aes(ymin = lower, ymax = upper), width = .2)
```

Figure 10-30 shows the resulting bar chart with confidence intervals.

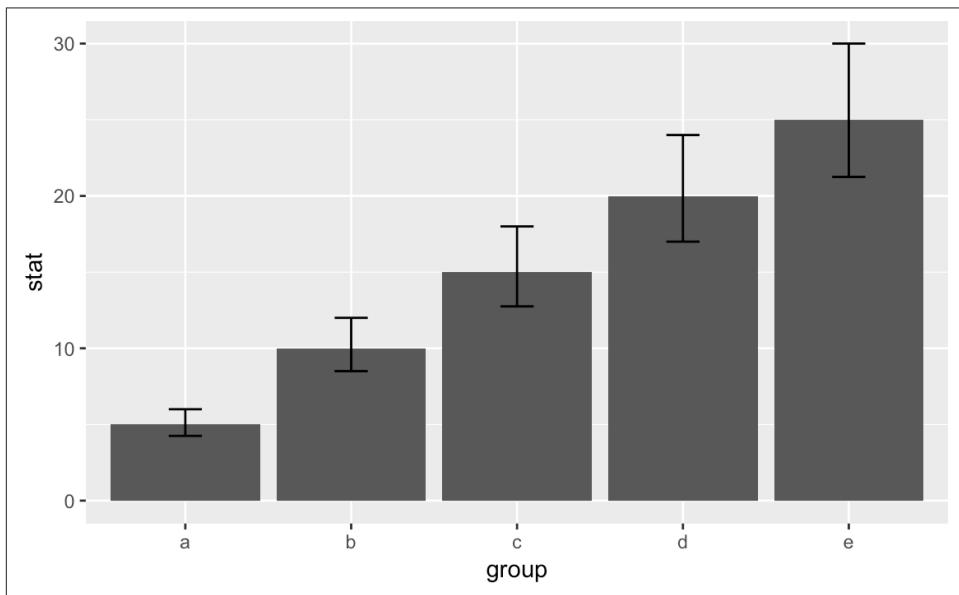


Figure 10-30. Bar chart with confidence intervals

Discussion

Most bar charts display point estimates, which are shown by the heights of the bars, but rarely do they include confidence intervals. Our inner statisticians dislike this intensely. The point estimate is only half of the story; the confidence interval gives the full story.

Fortunately, we can plot the error bars using `ggplot`. The hard part is calculating the intervals. In the previous examples our data had a simple -15% and $+20\%$ interval. However, in [Recipe 10.10](#) we calculated group means before plotting them. If we let `ggplot` do the calculations for us, we can use the built-in `mean_se` along with the `stat_summary` function to get the standard errors of the mean measures.

Let's use the `airquality` data we used previously. First we'll do the sorted factor procedure (from the prior recipe) to get the month names in the desired order:

```
aq_data <- airquality %>%  
  arrange(Month) %>%  
  mutate(month_abb = fct_inorder(month.abb[Month]))
```

Now we can plot the bars along with the associated standard errors, as in Figure 10-31:

```
ggplot(aq_data, aes(month_abb, Temp)) +  
  geom_bar(stat = "summary",  
           fun.y = "mean",  
           fill = "cornflowerblue") +  
  stat_summary(fun.data = mean_se, geom = "errorbar") +  
  labs(title = "Mean Temp by Month",  
       x = "",  
       y = "Temp (deg. F)")
```

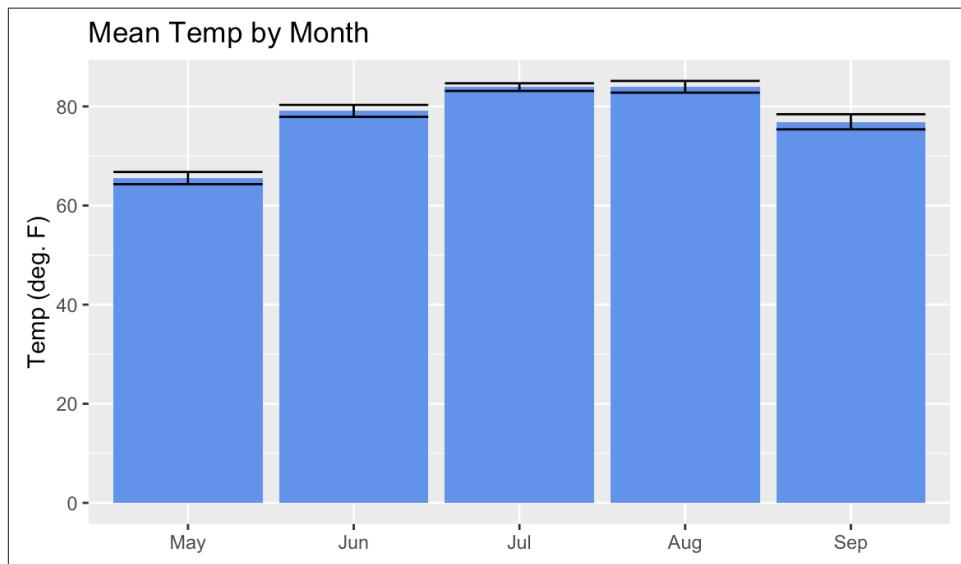


Figure 10-31. Mean temp by month with error bars

Sometimes you'll want to sort the columns in your bar chart in descending order based on their height, as in Figure 10-32. This can be a little bit confusing when you're using summary stats in ggplot, but the secret is to use `mean` in the `reorder` statement to sort the factor by the mean of the temp. Note that the reference to `mean` in `reorder` is not quoted, while the reference to `mean` in `geom_bar` is quoted:

```
ggplot(aq_data, aes(reorder(month_abb, -Temp, mean), Temp)) +  
  geom_bar(stat = "summary",  
           fun.y = "mean",  
           fill = "tomato") +  
  stat_summary(fun.data = mean_se, geom = "errorbar") +  
  labs(title = "Mean Temp by Month",  
       x = "",  
       y = "Temp (deg. F)")
```

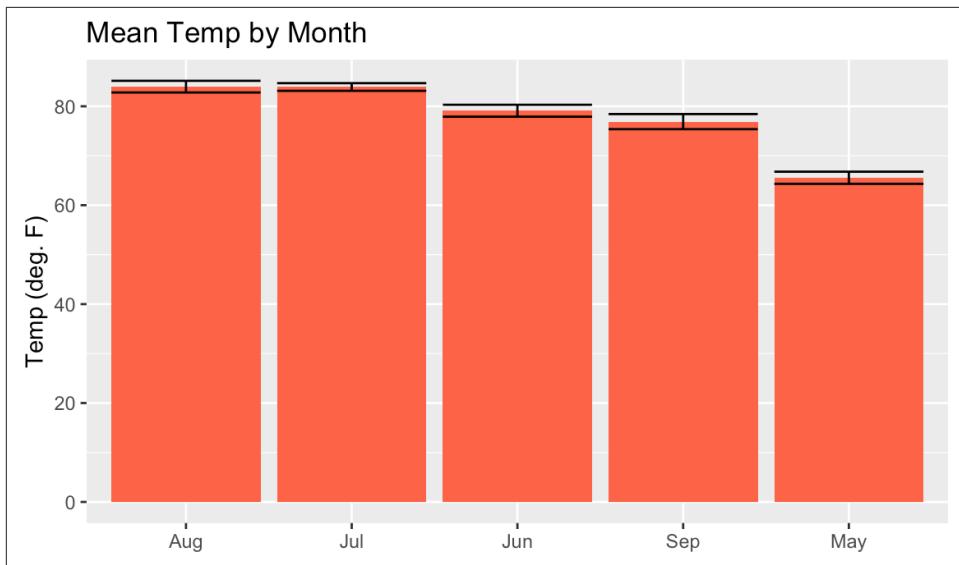


Figure 10-32. Mean temp by month in descending order

You may look at this example and the result in Figure 10-32 and wonder, “Why didn’t they just use `reorder(month_abb, Month)` in the first example instead of that sorting business with `forcats::fct_inorder` to get the months in the right order?” Well, we could have. But sorting using `fct_inorder` is a design pattern that provides flexibility for more complicated things. Plus it’s quite easy to read in a script. Using `reorder` inside `aes` is a bit denser and harder to read later, but either approach is reasonable.

See Also

See [Recipe 9.9](#) for more about `t.test`.

10.12 Coloring a Bar Chart

Problem

You want to color or shade the bars of a bar chart.

Solution

With `gplot` we add the `fill` parameter to our `aes` call and let `ggplot` pick the colors for us:

```
ggplot(df, aes(x, y, fill = group))
```

Discussion

We can use the `fill` parameter in `aes` to tell `ggplot` what field to base the colors on. If we pass a numeric field to `ggplot`, we will get a continuous gradient of colors, and if we pass a factor or character field to `fill`, we will get contrasting colors for each group. Here we pass the character name of each month to the `fill` parameter:

```
aq_data <- airquality %>%
  arrange(Month) %>%
  mutate(month_abb = fct_inorder(month.abb[Month]))  
  
ggplot(data = aq_data, aes(month_abb, Temp, fill = month_abb)) +
  geom_bar(stat = "summary", fun.y = "mean") +
  labs(title = "Mean Temp by Month",
       x = "",
       y = "Temp (deg. F)") +
  scale_fill_brewer(palette = "Paired")
```

We define the colors in the resulting bar chart (Figure 10-33) by calling `scale_fill_brewer(palette="Paired")`. The "Paired" color palette comes, along with many other color palettes, in the package `RColorBrewer`.

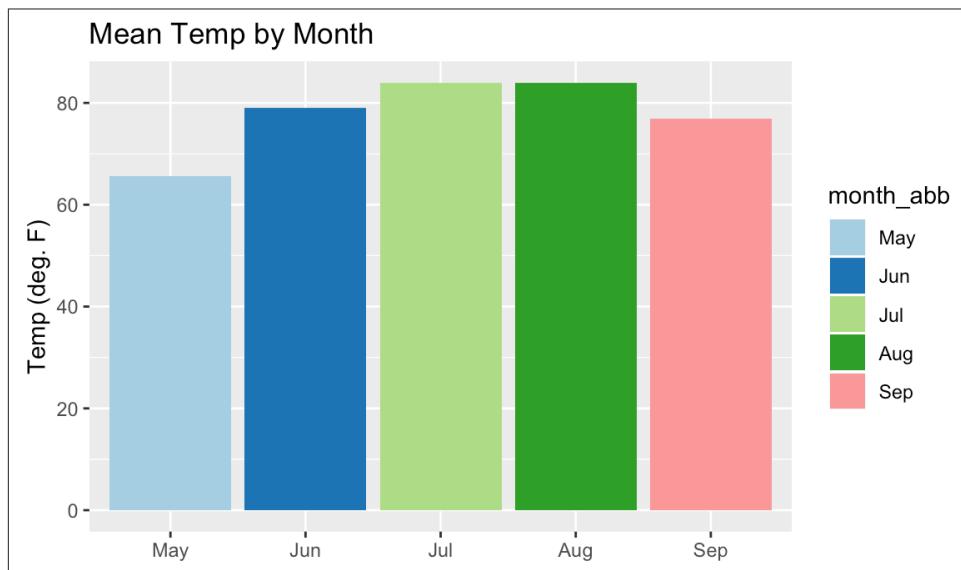


Figure 10-33. Colored monthly temp bar chart

If we want to change the color of each bar based on the temperature, we can't just set `fill = Temp`—as might seem intuitive—because `ggplot` won't understand we want the mean temperature after the grouping by month. The way we get around this is by accessing a special field inside of our graph called `..y...`, which is the calculated value on the y-axis. But we don't want the legend labeled `..y...`, so we add `fill = "Temp"`

to our `labs` call in order to change the name of the legend. The result is shown in Figure 10-34:

```
ggplot(airquality, aes(month.abb[Month], Temp, fill = ...y...)) +  
  geom_bar(stat = "summary", fun.y = "mean") +  
  labs(title = "Mean Temp by Month",  
       x = "",  
       y = "Temp (deg. F)",  
       fill = "Temp")
```

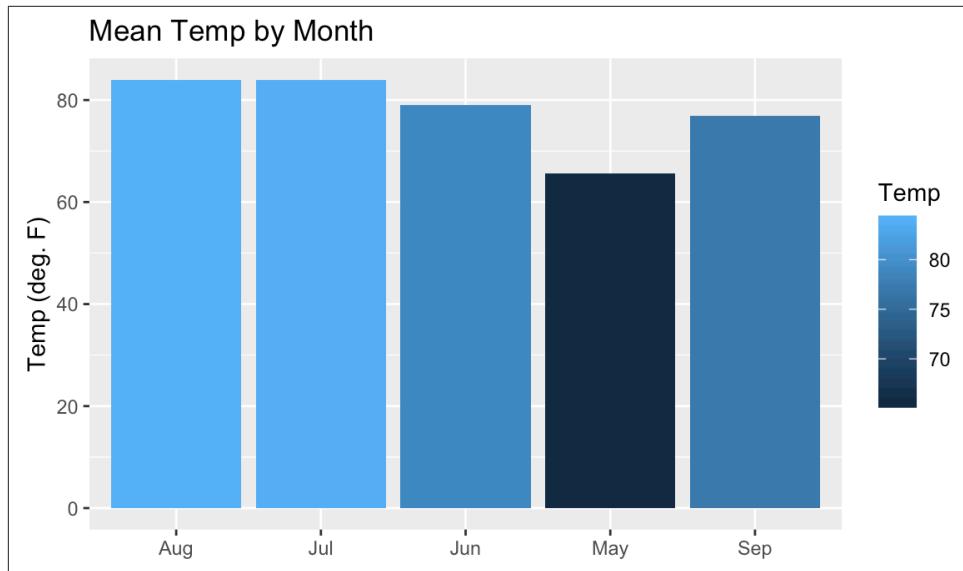


Figure 10-34. Bar chart shaded by value

If we want to reverse the color scale, we can just add a negative sign, `-`, in front of the field we are filling by: `fill=-...y...`, for example.

See Also

See [Recipe 10.10](#) for creating a bar chart.

10.13 Plotting a Line from x and y Points

Problem

You have paired observations in a data frame: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. You want to plot a series of line segments that connect the data points.

Solution

With `ggplot` we can use `geom_point` to plot the points:

```
ggplot(df, aes(x, y)) +  
  geom_point()
```

Since `ggplot` graphics are built up element by element, we can have both a point and a line in the same graphic very easily by having two geoms:

```
ggplot(df, aes(x , y)) +  
  geom_point() +  
  geom_line()
```

Discussion

To illustrate, let's look at some example US economic data that comes with `ggplot2`. This example data frame has a column called `date`, which we'll plot on the x-axis, and a field called `unemploy`, which is the number of unemployed people:

```
ggplot(economics, aes(date , unemploy)) +  
  geom_point() +  
  geom_line()
```

Figure 10-35 shows the resulting chart, which contains both lines and points because we used both geoms.

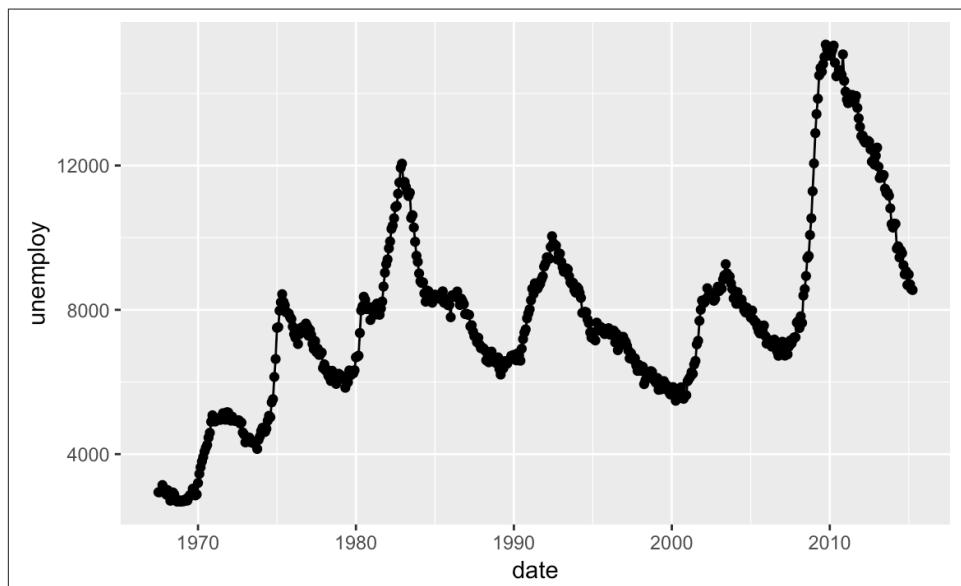


Figure 10-35. Line chart

See Also

See [Recipe 10.1](#).

10.14 Changing the Type, Width, or Color of a Line

Problem

You are plotting a line, and you want to change its type, width, or color.

Solution

`ggplot` uses the `linetype` parameter for controlling the appearance of lines. The options are:

- `linetype="solid"` or `linetype=1` (default)
- `linetype="dashed"` or `linetype=2`
- `linetype="dotted"` or `linetype=3`
- `linetype="dotdash"` or `linetype=4`
- `linetype="longdash"` or `linetype=5`
- `linetype="twodash"` or `linetype=6`
- `linetype="blank"` or `linetype=0` (inhibits drawing)

We can change the line characteristics by passing `linetype`, `col`, and/or `size` as parameters to `geom_line`. For example, if we wanted to change the line type to dashed, red, and heavy, we could pass the following params to `geom_line`:

```
ggplot(df, aes(x, y)) +  
  geom_line(linetype = 2,  
            size = 2,  
            col = "red")
```

Discussion

The example syntax shows how to draw one line and specify its style, width, or color. A common scenario involves drawing multiple lines, each with its own style, width, or color.

In `ggplot` this can be a conundrum for many users. The challenge is that `ggplot` works best with “long” data instead of “wide” data, as was mentioned in the introduction to this chapter.

Let's set up some example data:

```
x <- 1:10
y1 <- x**1.5
y2 <- x**2
y3 <- x**2.5
df <- data.frame(x, y1, y2, y3)
```

Our example data frame has four columns of wide data:

```
head(df, 3)
#>   x   y1  y2   y3
#> 1 1  1.00 1  1.00
#> 2 2  2.83 4  5.66
#> 3 3  5.20 9 15.59
```

We can make our wide data long by using the `gather` function from the core tidyverse package `tidyverse`. In this example, we use `gather` to create a new column named `bucket` and put our column names in there while keeping our `x` and `y` variables:

```
df_long <- gather(df, bucket, y, -x)
head(df_long, 3)
#>   x bucket   y
#> 1 1     y1 1.00
#> 2 2     y1 2.83
#> 3 3     y1 5.20
tail(df_long, 3)
#>   x bucket   y
#> 28 8     y3 181
#> 29 9     y3 243
#> 30 10    y3 316
```

Now we can pass `bucket` to the `col` parameter and get multiple lines, each a different color:

```
ggplot(df_long, aes(x, y, col = bucket)) +
  geom_line()
```

Figure 10-36 shows the resulting graph with each variable represented in a different color.

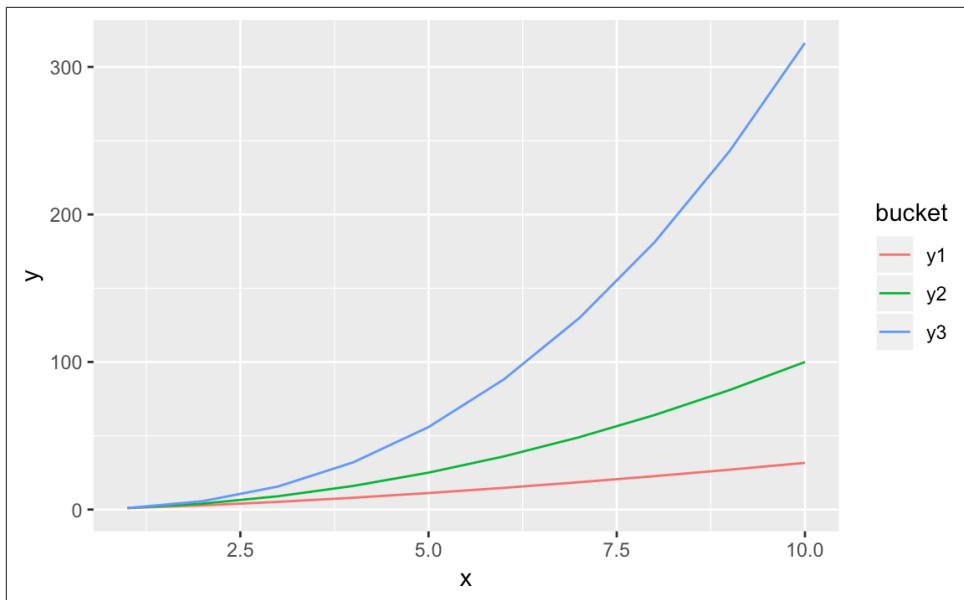


Figure 10-36. Multiple line chart

It's straightforward to vary the line weight by a variable—simply pass a numerical variable to `size`:

```
ggplot(df, aes(x, y1, size = y2)) +  
  geom_line() +  
  scale_size(name = "Thickness based on y2")
```

The result of varying the thickness with x is shown in [Figure 10-37](#).

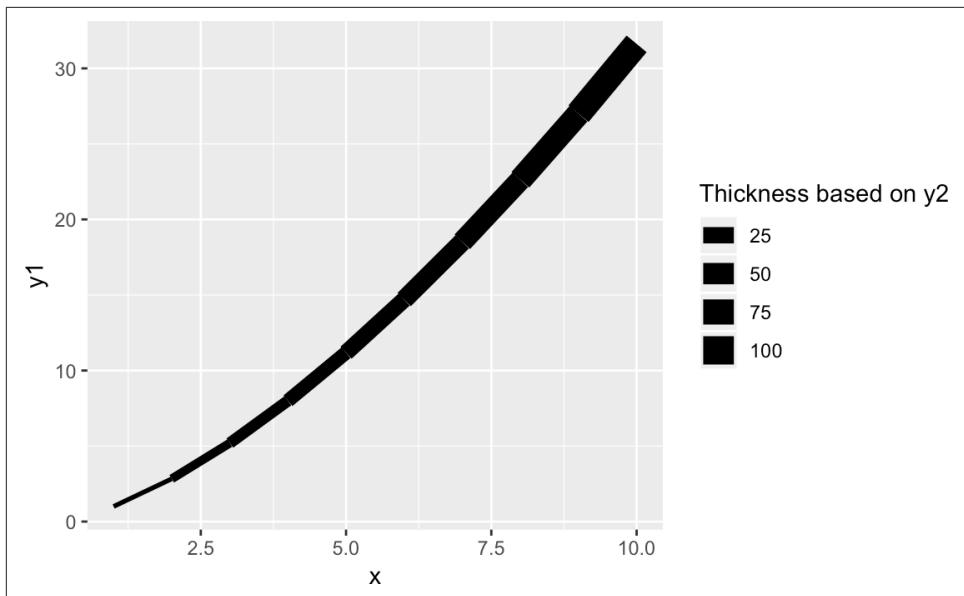


Figure 10-37. Thickness as a function of x

See Also

See [Recipe 10.13](#) for plotting a basic line.

10.15 Plotting Multiple Datasets

Problem

You want to show multiple datasets in one plot.

Solution

We can add multiple data frames to a `ggplot` figure by creating an empty plot and then adding two different geoms to the plot:

```
ggplot() +  
  geom_line(data = df1, aes(x1, y1)) +  
  geom_line(data = df2, aes(x2, y2))
```

This code uses `geom_line`, but you could use any geom.

Discussion

We could combine the data into one data frame before plotting using one of the join functions from `dplyr`. However, next we will create two separate data frames and then add them each to a `ggplot` graph.

First let's set up our example data frames, `df1` and `df2`:

```
# example data
n <- 20

x1 <- 1:n
y1 <- rnorm(n, 0, .5)
df1 <- data.frame(x1, y1)

x2 <- (.5 * n):(1.5 * n) - 1
y2 <- rnorm(n, 1, .5)
df2 <- data.frame(x2, y2)
```

Typically we would pass the data frame directly into the `ggplot` function call. Since we want two geoms with two different data sources, we will initiate a plot with `ggplot` and then add in two calls to `geom_line`, each with its own data source:

```
ggplot() +
  geom_line(data = df1, aes(x1, y1), color = "darkblue") +
  geom_line(data = df2, aes(x2, y2), linetype = "dashed")
```

`ggplot` allows us to make multiple calls to different `geom_` functions, each with its own data source, if desired. Then `ggplot` will look at all the data we are plotting and adjust the ranges to accommodate all the data.

The graph with expanded limits is shown in [Figure 10-38](#).

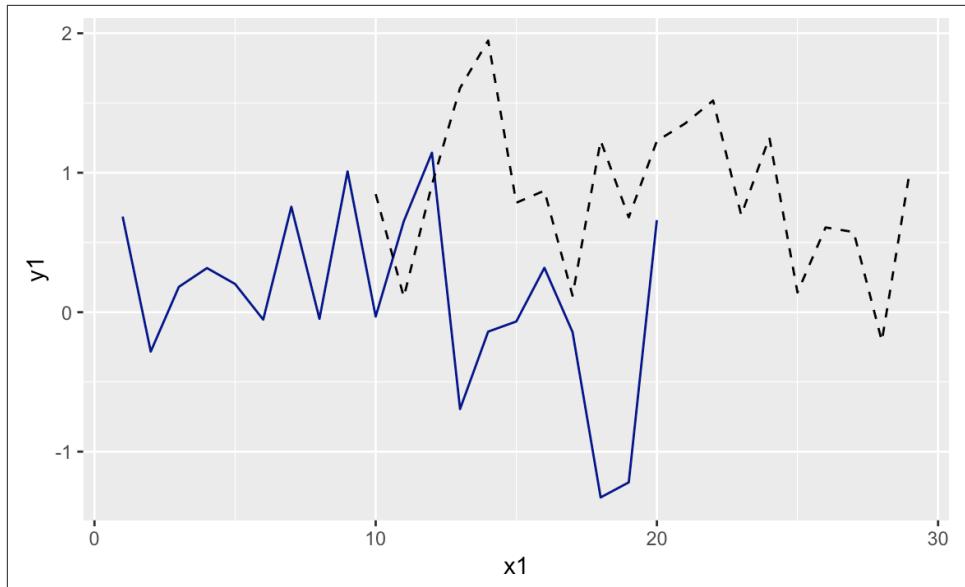


Figure 10-38. Two lines, one plot

10.16 Adding Vertical or Horizontal Lines

Problem

You want to add a vertical or horizontal line to your plot, such as an axis through the origin or a pointer to a threshold.

Solution

The `ggplot` functions `geom_vline` and `geom_hline` produce vertical and horizontal lines, respectively. The functions can also take `color`, `linetype`, and `size` parameters to set the line style:

```
# using the data.frame df1 from the prior recipe
ggplot(df1) +
  aes(x = x1, y = y1) +
  geom_point() +
  geom_vline(
    xintercept = 10,
    color = "red",
    linetype = "dashed",
    size = 1.5
  ) +
  geom_hline(yintercept = 0, color = "blue")
```

Figure 10-39 shows the resulting plot with added horizontal and vertical lines.

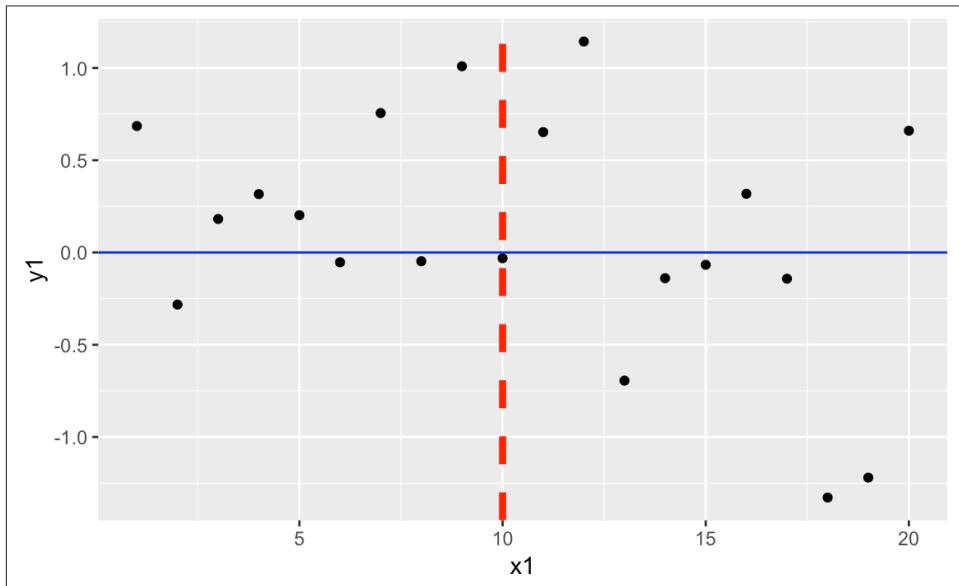


Figure 10-39. Vertical and horizontal lines

Discussion

A typical use of lines would be drawing regularly spaced lines. Suppose we have a sample of points, samp. First, we plot them with a solid line through the mean. Then we calculate and draw dotted lines at ± 1 and ± 2 standard deviations away from the mean. We can add the lines into our plot with geom_hline:

```
samp <- rnorm(1000)
samp_df <- data.frame(samp, x = 1:length(samp))

mean_line <- mean(samp_df$samp)
sd_lines <- mean_line + c(-2, -1, +1, +2) * sd(samp_df$samp)

ggplot(samp_df) +
  aes(x = x, y = samp) +
  geom_point() +
  geom_hline(yintercept = mean_line, color = "darkblue") +
  geom_hline(yintercept = sd_lines, linetype = "dotted")
```

Figure 10-40 shows the sampled data along with the mean and standard deviation lines.

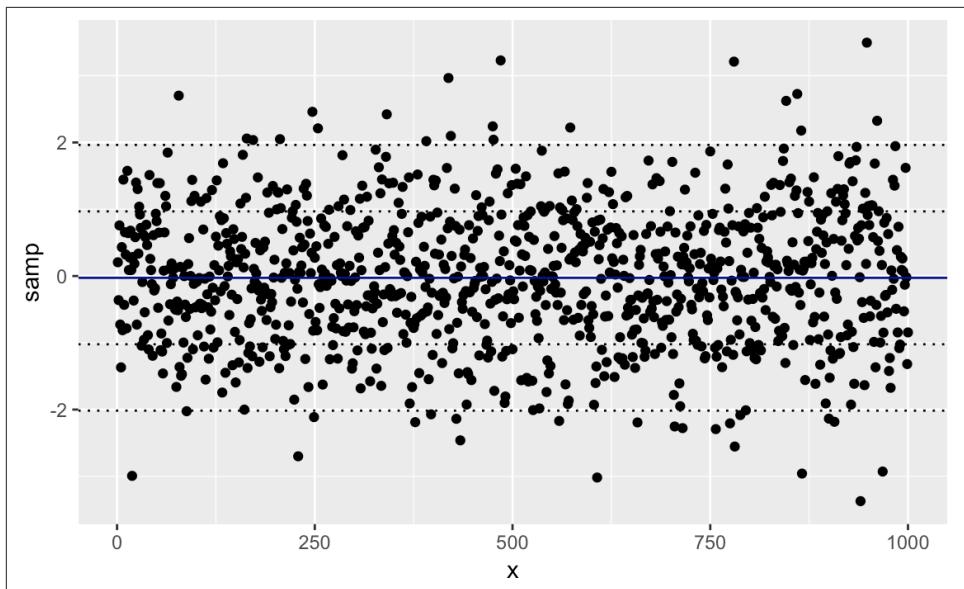


Figure 10-40. Mean and SD bands in a plot

See Also

See [Recipe 10.14](#) for more about changing line types.

10.17 Creating a Boxplot

Problem

You want to create a boxplot of your data.

Solution

Use `geom_boxplot` from `ggplot` to add a boxplot geom to a `ggplot` graphic. Using the `samp_df` data frame from the prior recipe, we can create a boxplot of the values in the `x` column. The resulting graph is shown in [Figure 10-41](#):

```
ggplot(samp_df) +  
  aes(y = samp) +  
  geom_boxplot()
```

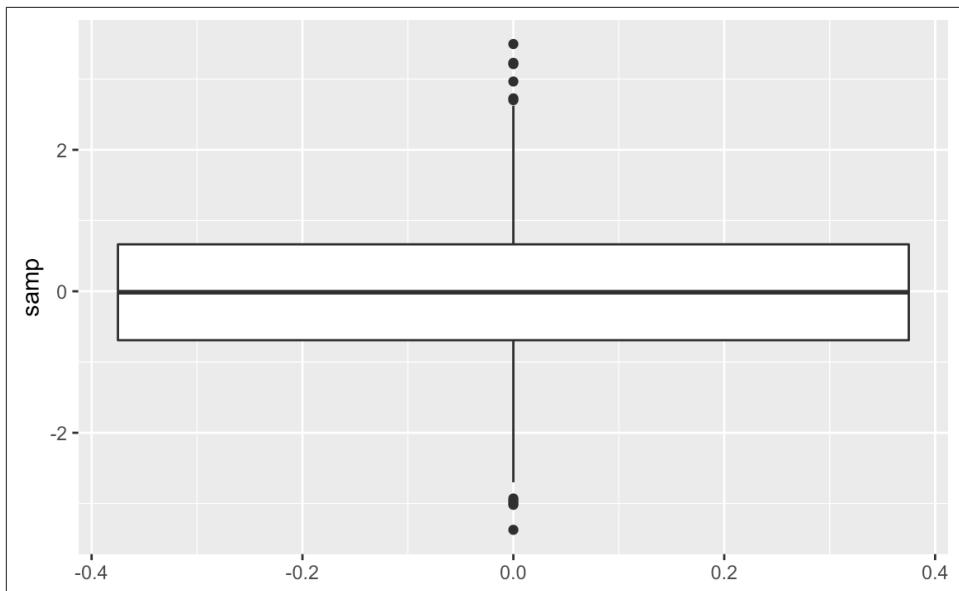


Figure 10-41. Single boxplot

Discussion

A boxplot provides a quick and easy visual summary of a dataset:

- The thick line in the middle is the median.
- The box surrounding the median identifies the first and third quartiles; the bottom of the box is Q1, and the top is Q3.
- The “whiskers” above and below the box show the range of the data, excluding outliers.
- The circles identify outliers. By default, an outlier is defined as any value that is farther than $1.5 \times \text{IQR}$ away from the box. (IQR is the *interquartile range*, or $\text{Q}_3 - \text{Q}_1$.) In this example, there are a few outliers on the high side.

We can rotate the boxplot by flipping the coordinates. There are some situations where this makes a more appealing graphic, as shown in [Figure 10-42](#):

```
ggplot(samp_df) +  
  aes(y = samp) +  
  geom_boxplot() +  
  coord_flip()
```

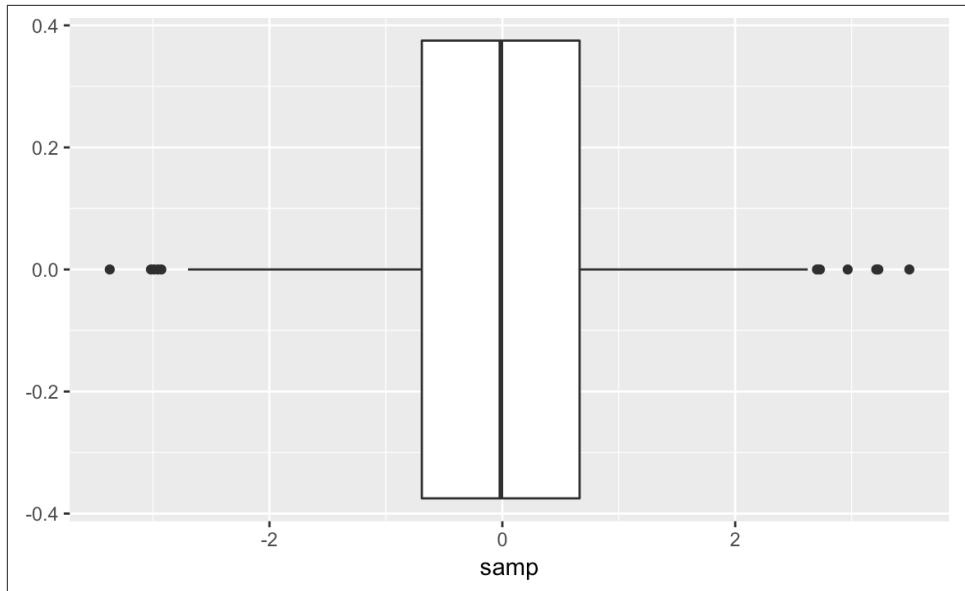


Figure 10-42. Single boxplot, flipped

See Also

One boxplot alone is pretty boring. See [Recipe 10.18](#) for creating multiple boxplots.

10.18 Creating One Boxplot for Each Factor Level

Problem

Your dataset contains a numeric variable and a factor (or other categorical text). You want to create several boxplots of the numeric variable broken out by levels.

Solution

With `ggplot` we pass the name of the categorical variable to the `x` parameter in the `aes` call. The resulting boxplot will then be grouped by the values in the categorical variable:

```
ggplot(df) +  
  aes(x = factor, y = values) +  
  geom_boxplot()
```

Discussion

This recipe is another great way to explore and illustrate the relationship between two variables. In this case, we want to know whether the numeric variable changes according to the level of a category.

The `UScereal` dataset from the `MASS` package contains many variables regarding breakfast cereals. One variable is the amount of sugar per portion and another is the shelf position (counting from the floor). Cereal manufacturers can negotiate for shelf position, placing their products for the best sales potential. We wonder: where do they put the high-sugar cereals? We can produce [Figure 10-43](#) and explore that question by creating one boxplot per shelf:

```
data(UScereal, package = "MASS")  
  
ggplot(UScereal) +  
  aes(x = as.factor(shelf), y = sugars) +  
  geom_boxplot() +  
  labs(  
    title = "Sugar Content by Shelf",  
    x = "Shelf",  
    y = "Sugar (grams per portion)"  
)
```

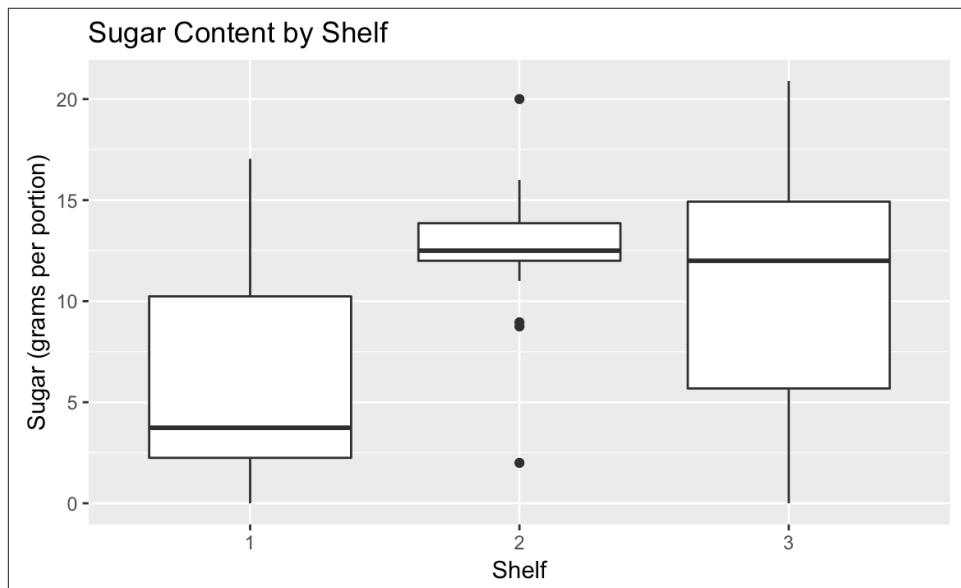


Figure 10-43. Boxplots by shelf number

The boxplots suggest that shelf #2 has the most high-sugar cereals. Could it be that this shelf is at eye level for young children who can influence their parents' choice of cereals?



Note that in the `aes` call we had to tell `ggplot` to treat the shelf number as a factor. Otherwise, `ggplot` would not react to the shelf as a grouping and would print only a single boxplot.

See Also

See [Recipe 10.17](#) for creating a basic boxplot.

10.19 Creating a Histogram

Problem

You want to create a histogram of your data.

Solution

Use `geom_histogram`, and set `x` to a vector of numeric values.

Discussion

[Figure 10-44](#) is a histogram of the `MPG.city` column taken from the `Cars93` dataset:

```
data(Cars93, package = "MASS")  
  
ggplot(Cars93) +  
  geom_histogram(aes(x = MPG.city))  
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

The `geom_histogram` function must decide how many cells (bins) to create for binning the data. In this example, the default algorithm chose 30 bins. If we wanted fewer bins, we would include the `bins` parameter to tell `geom_histogram` how many bins we want:

```
ggplot(Cars93) +  
  geom_histogram(aes(x = MPG.city), bins = 13)
```

[Figure 10-45](#) shows the histogram with 13 bins.

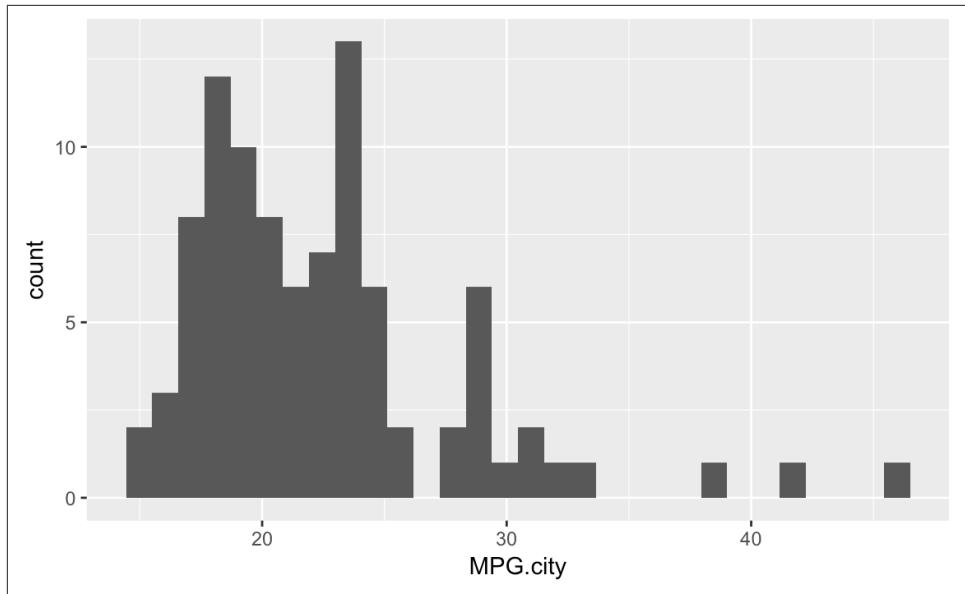


Figure 10-44. Histogram of counts by MPG

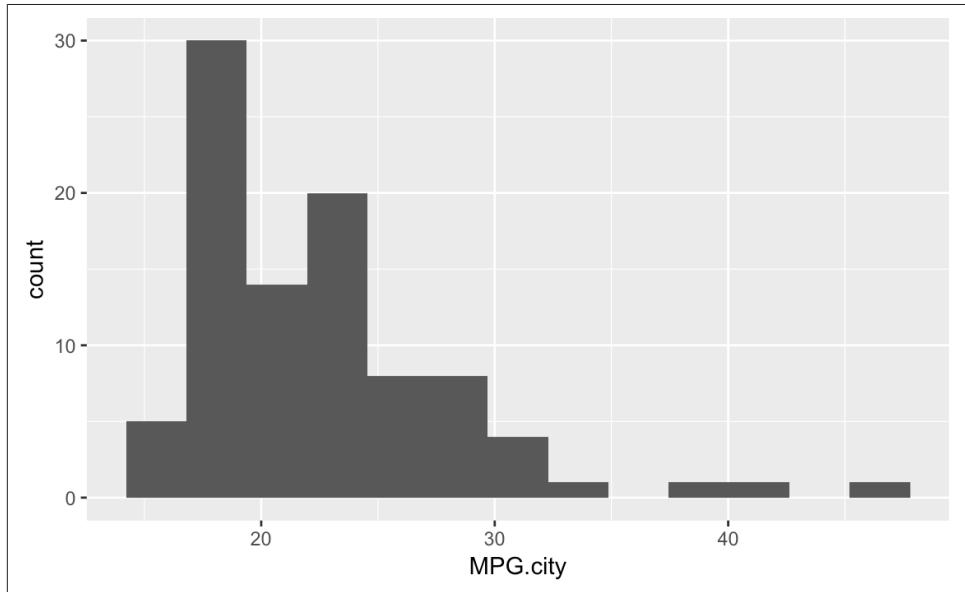


Figure 10-45. Histogram of counts by MPG with fewer bins

See Also

The Base R function `hist` provides much of the same functionality, as does the `histogram` function of the `lattice` package.

10.20 Adding a Density Estimate to a Histogram

Problem

You have a histogram of your data sample, and you want to add a curve to illustrate the apparent density.

Solution

Use the `geom_density` function to approximate the sample density, as shown in Figure 10-46:

```
ggplot(Cars93) +  
  aes(x = MPG.city) +  
  geom_histogram(aes(y = ..density..), bins = 21) +  
  geom_density()
```

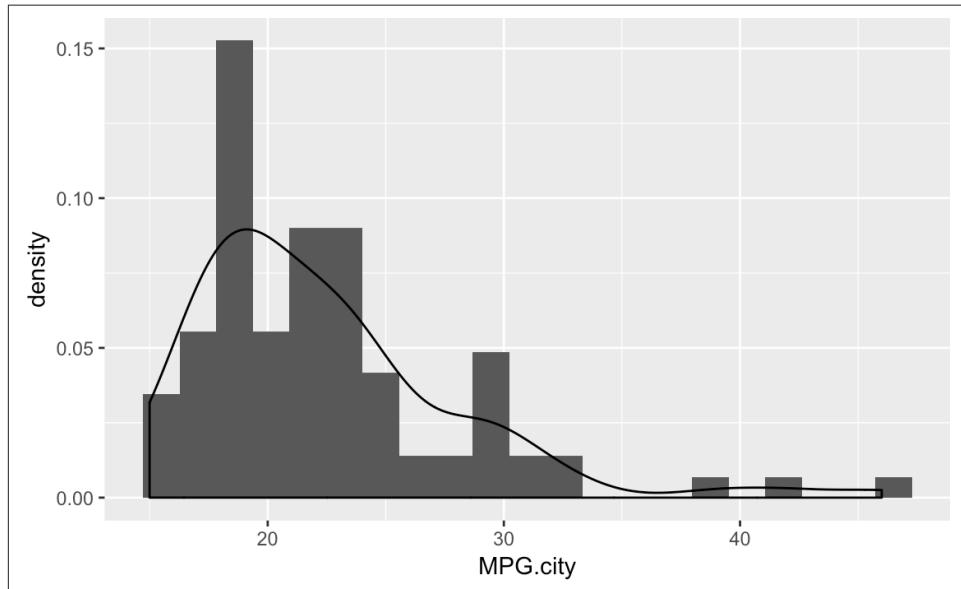


Figure 10-46. Histogram with density plot

Discussion

A histogram suggests the density function of your data, but it is rough. A smoother estimate could help you better visualize the underlying distribution. A *kernel density estimation* (KDE) is a smoother representation of univariate data.

In `ggplot` we tell the `geom_histogram` function to use the `geom_density` function by passing it `aes(y = ..density..)`.

The following example takes a sample from a gamma distribution and then plots the histogram and the estimated density, as shown in [Figure 10-47](#):

```
samp <- rgamma(500, 2, 2)

ggplot() +
  aes(x = samp) +
  geom_histogram(aes(y = ..density..), bins = 10) +
  geom_density()
```

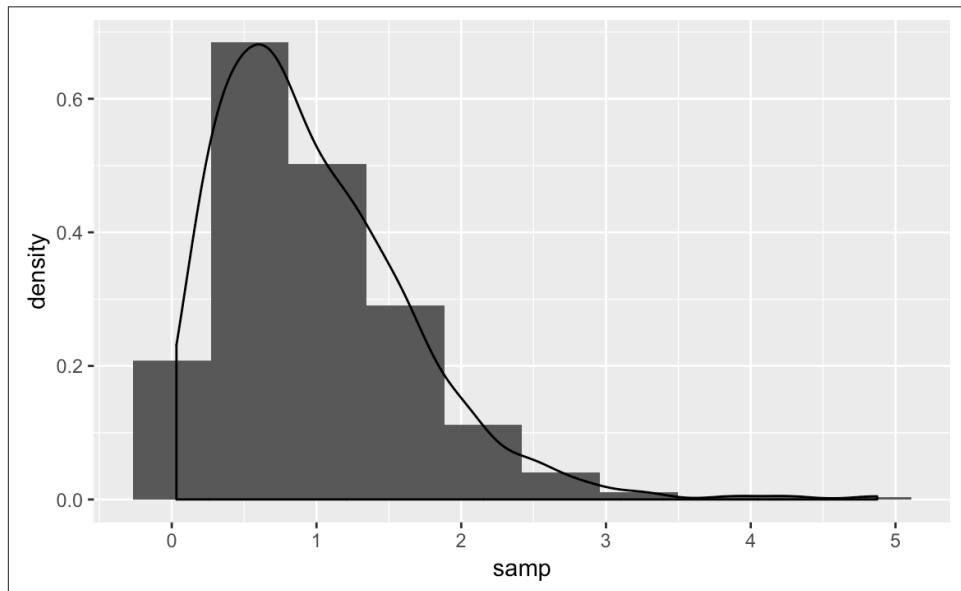


Figure 10-47. Histogram and density: gamma distribution

See Also

The `geom_density` function approximates the shape of the density nonparametrically. If you know the actual underlying distribution, use [Recipe 8.11](#) to plot the density function instead.

10.21 Creating a Normal Quantile–Quantile Plot

Problem

You want to create a *quantile–quantile* (Q–Q) plot of your data, typically because you want to know how the data differs from a normal distribution.

Solution

With `ggplot` we can use the `stat_qq` and `stat_qq_line` functions to create a Q–Q plot that shows the observed points as well as the Q–Q line. [Figure 10-48](#) shows the resulting plot:

```
df <- data.frame(x = rnorm(100))

ggplot(df, aes(sample = x)) +
  stat_qq() +
  stat_qq_line()
```

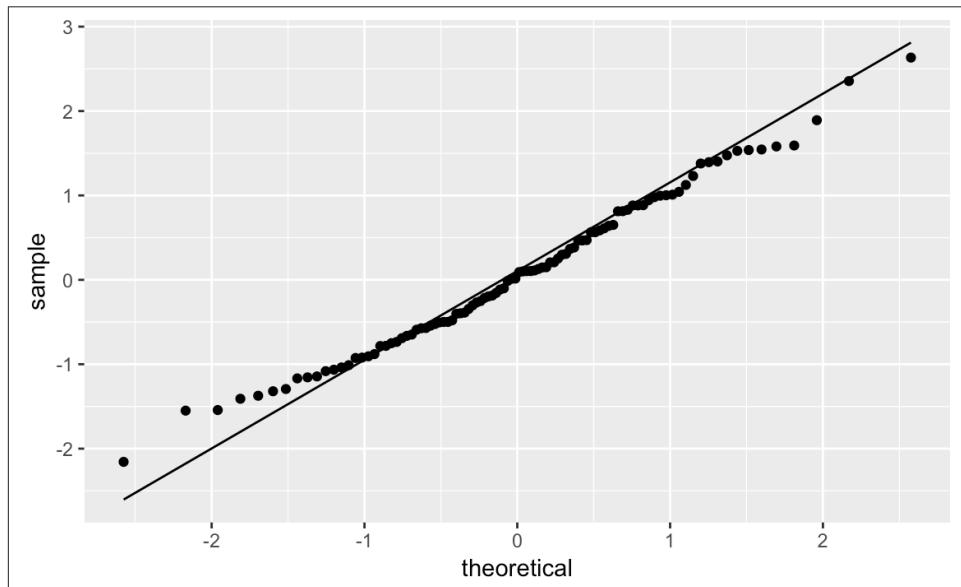


Figure 10-48. Q–Q plot

Discussion

Sometimes it's important to know if your data is normally distributed. A quantile–quantile (Q–Q) plot is a good first check.

The `Cars93` dataset contains a `Price` column. Is it normally distributed? This code snippet creates a Q–Q plot of `Price`, as shown in [Figure 10-49](#):

```
ggplot(Cars93, aes(sample = Price)) +  
  stat_qq() +  
  stat_qq_line()
```

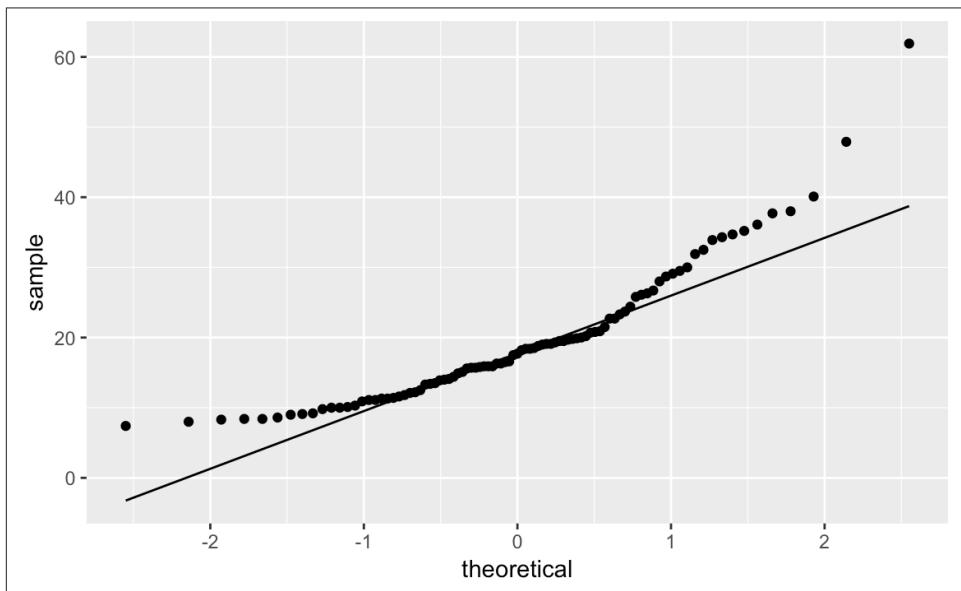


Figure 10-49. Q-Q plot of car prices

If the data had a perfect normal distribution, then the points would fall exactly on the diagonal line. Many points are close, especially in the middle section, but the points in the tails are pretty far off. Too many points are above the line, indicating a general skew to the left.

The leftward skew might be cured by a logarithmic transformation. We can plot `log(Price)`, which yields Figure 10-50:

```
ggplot(Cars93, aes(sample = log(Price))) +  
  stat_qq() +  
  stat_qq_line()
```

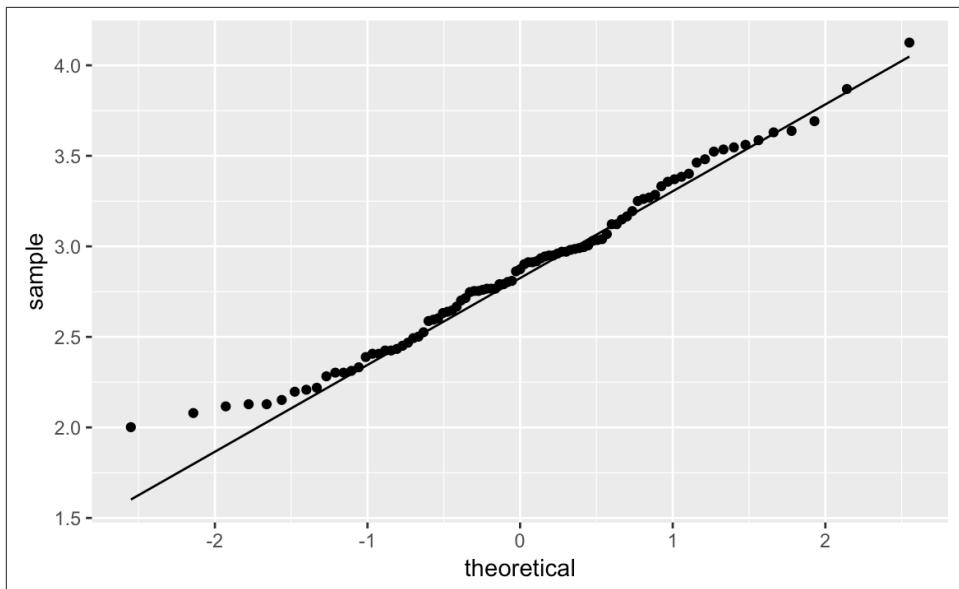


Figure 10-50. Q-Q plot of log car prices

Notice that the points in the new plot are much better behaved, staying close to the line except in the extreme left tail. It appears that `log(Price)` is approximately normal.

See Also

See [Recipe 10.22](#) for creating Q–Q plots for other distributions. See [Recipe 11.16](#) for an application of Normal Q–Q plots to diagnose linear regression.

10.22 Creating Other Quantile–Quantile Plots

Problem

You want to view a quantile-quantile plot for your data, but the data is not normally distributed.

Solution

For this recipe, you must have some idea of the underlying distribution, of course. The solution is built from the following steps:

1. Use the `ppoints` function to generate a sequence of points between 0 and 1.

2. Transform those points into quantiles, using the quantile function for the assumed distribution.
3. Sort your sample data.
4. Plot the sorted data against the computed quantiles.
5. Use `abline` to plot the diagonal line.

This can all be done in two lines of R code. Here is an example that assumes your data, `y`, has a Student's t distribution with 5 degrees of freedom. Recall that the quantile function for Student's t is `qt` and that its second argument is the degrees of freedom.

First let's make some example data:

```
df_t <- data.frame(y = rt(100, 5))
```

In order to create the Q-Q plot we need to estimate the parameters of the distribution we want to plot. Since this is a Student's t distribution, we only need to estimate one parameter, the degrees of freedom. Of course we know the actual degrees of freedom is 5, but in most situations we'll need to calculate that value. So, we'll use the `MASS::fitdistr` function to estimate the degrees of freedom:

```
est_df <- as.list(MASS::fitdistr(df_t$y, "t")$estimate)[["df"]]
est_df
#> [1] 19.5
```

As expected, that's pretty close to what was used to generate the simulated data, so let's pass the estimated degrees of freedom to the Q-Q functions and create [Figure 10-51](#):

```
ggplot(df_t) +
  aes(sample = y) +
  geom_qq(distribution = qt, dparams = est_df) +
  stat_qq_line(distribution = qt, dparams = est_df)
```

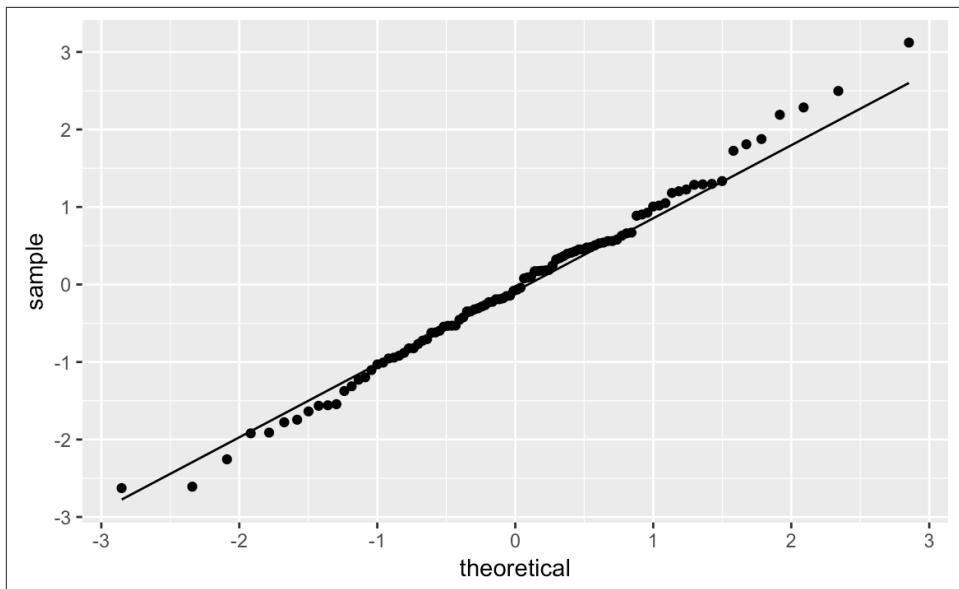


Figure 10-51. Student's t distribution Q-Q plot

Discussion

The Solution looks complicated, but the gist of it is picking a distribution, fitting the parameters, and then passing those parameters to the Q-Q functions in `ggplot`.

We can illustrate this recipe by taking a random sample from an exponential distribution with a mean of 10 (or, equivalently, a rate of 1/10):

```
rate <- 1 / 10
n <- 1000
df_exp <- data.frame(y = rexp(n, rate = rate))

est_exp <- as.list(MASS::fitdistr(df_exp$y, "exponential")$estimate)[["rate"]]
est_exp
#> [1] 0.101
```

Notice that for an exponential distribution, the parameter we estimate is called `rate` as opposed to `df`, which was the parameter in the t distribution.

The quantile function for the exponential distribution is `qexp`, which takes the `rate` argument. Figure 10-52 shows the resulting Q-Q plot using a theoretical exponential distribution:

```
ggplot(df_exp) +  
  aes(sample = y) +  
  geom_qq(distribution = qexp, dparams = est_exp) +  
  stat_qq_line(distribution = qexp, dparams = est_exp)
```

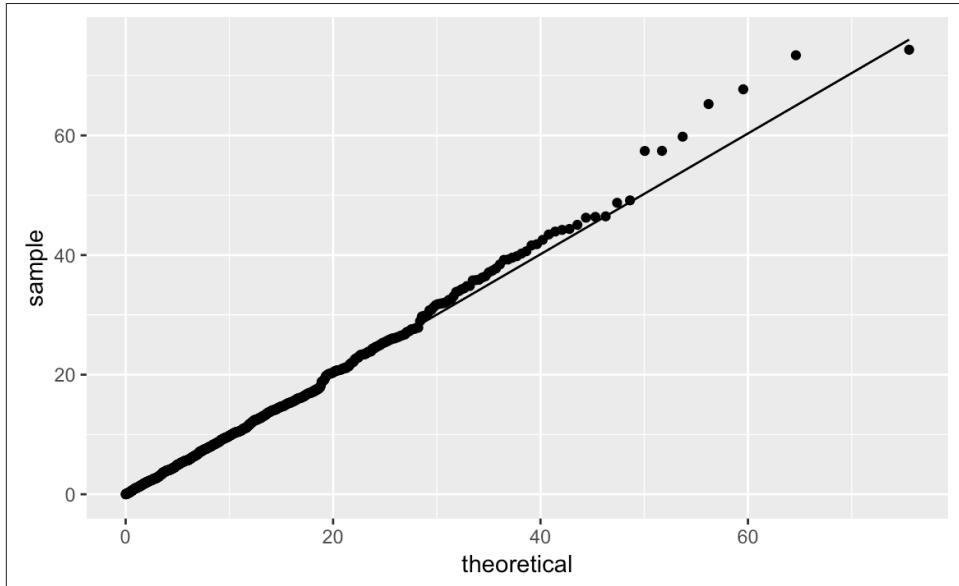


Figure 10-52. Exponential distribution Q-Q plot

10.23 Plotting a Variable in Multiple Colors

Problem

You want to plot your data in multiple colors, typically to make the plot more informative, readable, or interesting.

Solution

We can pass a color to a `geom_` function in order to produce colored output (see Figure 10-53):

```
df <- data.frame(x = rnorm(200), y = rnorm(200))  
  
ggplot(df) +  
  aes(x = x, y = y) +  
  geom_point(color = "blue")
```

If you are reading this in print you may see only black. Try it out on your own in order to see the graph in full color.

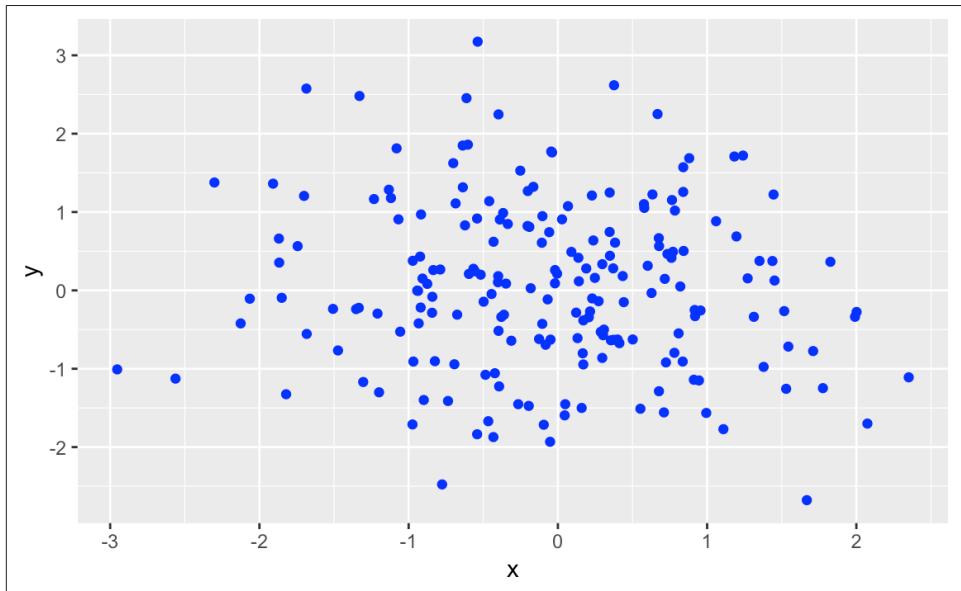


Figure 10-53. Point data in color

The value of `color` can be:

- One color, in which case all data points are that color.
- A vector of colors, the same length as `x`, in which case each value of `x` is colored with its corresponding color.
- A short vector, in which case the vector of colors is recycled.

Discussion

The default color in `ggplot` is black. While it's not very exciting, black is high contrast and easy for almost anyone to see.

However, it is much more useful (and interesting) to vary the color in a way that illuminates the data. Let's illustrate this by plotting a graphic two ways, once in black and white and once with simple shading.

This produces the basic black-and-white graphic in [Figure 10-54](#):

```
df <- data.frame(  
  x = 1:100,  
  y = rnorm(100))
```

```
)  
  
ggplot(df) +  
  aes(x, y) +  
  geom_point()
```

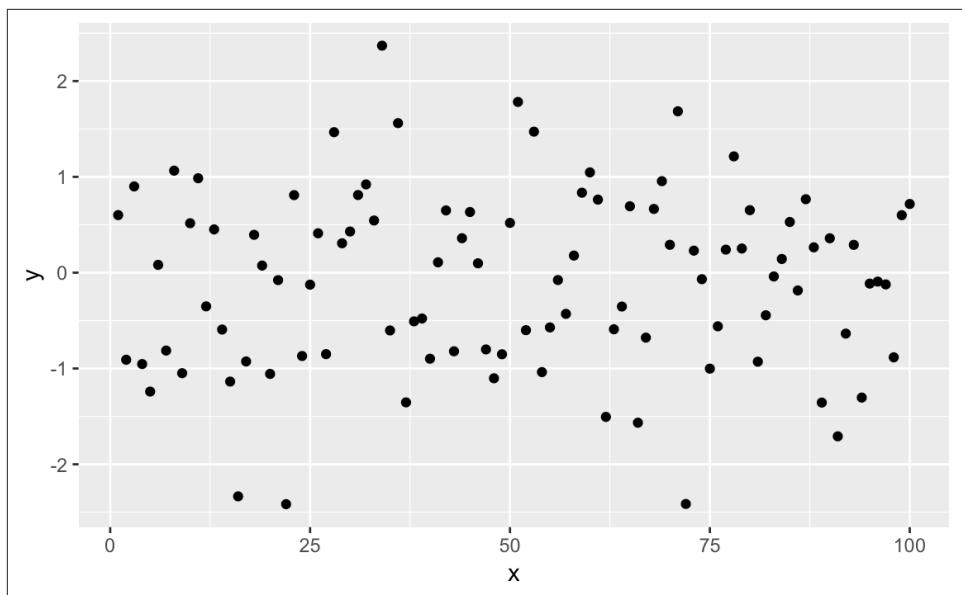


Figure 10-54. Simple point plot

Now we can make it more interesting by creating a vector of "gray" and "black" values, according to the sign of x, and then plotting x using those colors, as shown in Figure 10-55:

```
shade <- if_else(df$y >= 0, "black", "gray")  
  
ggplot(df) +  
  aes(x, y) +  
  geom_point(color = shade)
```

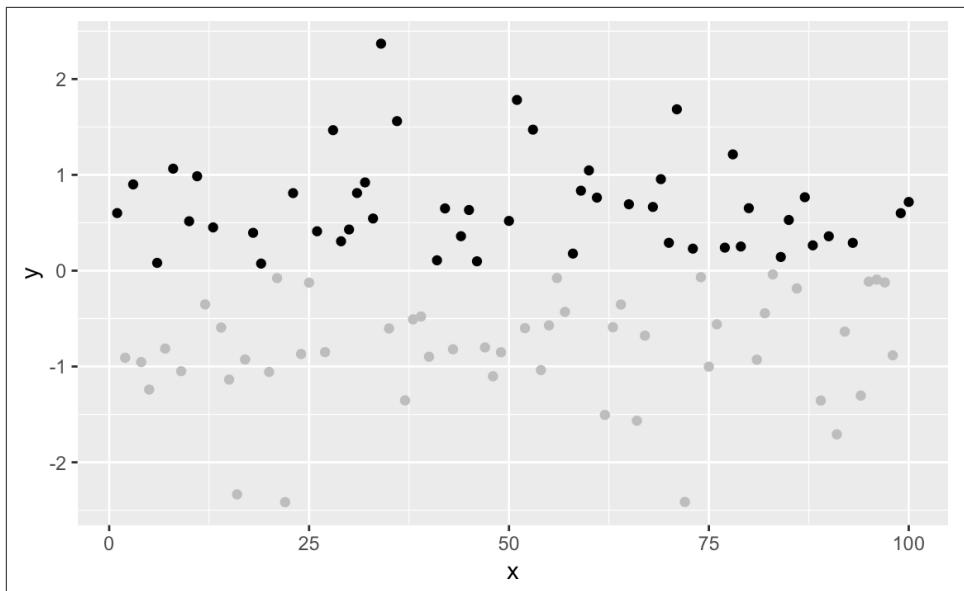


Figure 10-55. Color-shaded point plot

The negative values are now plotted in gray because the corresponding element of colors is "gray".

See Also

See [Recipe 5.3](#) regarding the Recycling Rule. Execute colors to see a list of available colors, and use geom_segment in ggplot to plot line segments in multiple colors.

10.24 Graphing a Function

Problem

You want to graph the value of a function.

Solution

The ggplot function stat_function will graph a function across a range. In [Figure 10-56](#), we plot a sine wave across the range -3 to 3:

```
ggplot(data.frame(x = c(-3, 3))) +  
  aes(x) +  
  stat_function(fun = sin)
```

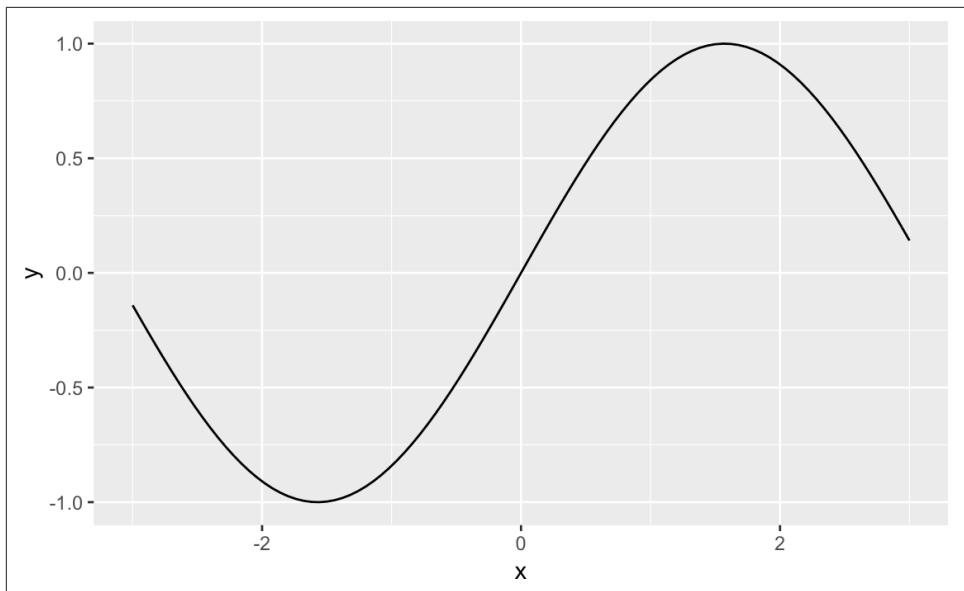


Figure 10-56. Sine wave plot

Discussion

It's pretty common to want to plot a statistical function, such as a normal distribution, across a given range. `stat_function` in `ggplot` allows us to do this. We need only supply a data frame with `x` value limits, and `stat_function` will calculate the `y` values and plot the results as shown in [Figure 10-57](#):

```
ggplot(data.frame(x = c(-3.5, 3.5))) +  
  aes(x) +  
  stat_function(fun = dnorm) +  
  ggtitle("Standard Normal Density")
```

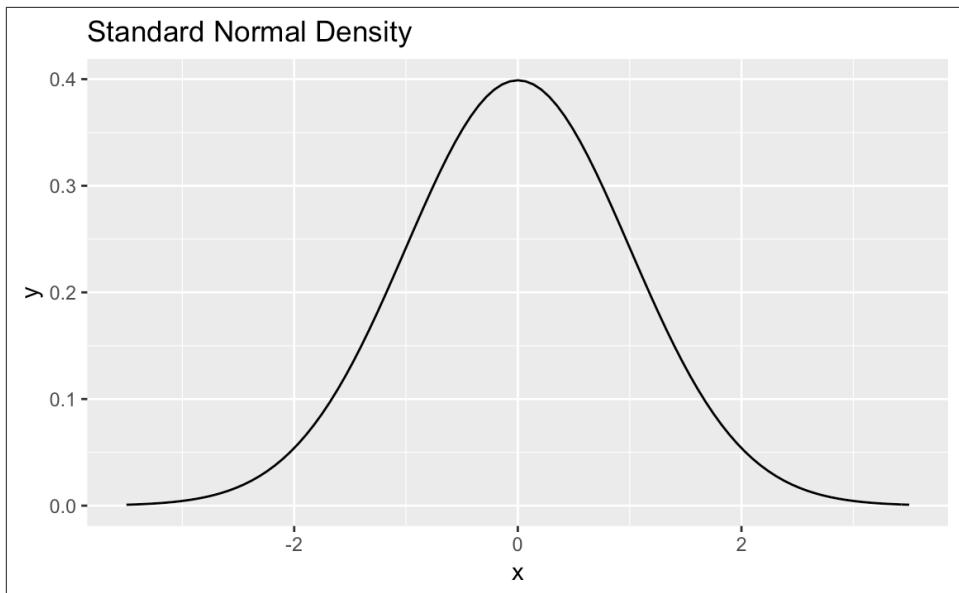


Figure 10-57. Standard Normal density plot

Notice here that we used `ggtitle` to set the title. If setting multiple text elements in a `ggplot` we use `labs`, but when we're just adding a title, `ggtitle` is more concise than `labs(title='Standard Normal Density')`, although they accomplish the same thing. See `?labs` for more discussion of labels with `ggplot`.

`stat_function` can graph any function that takes one argument and returns one value. Let's create a function and then plot it. Our function is a damped sine wave—that is, a sine wave that loses amplitude as it moves away from 0:

```
f <- function(x) exp(-abs(x)) * sin(2 * pi * x)
ggplot(data.frame(x = c(-3.5, 3.5))) +
  aes(x) +
  stat_function(fun = f) +
  ggtitle("Damped Sine Wave")
```

The resulting plot is shown in [Figure 10-58](#).

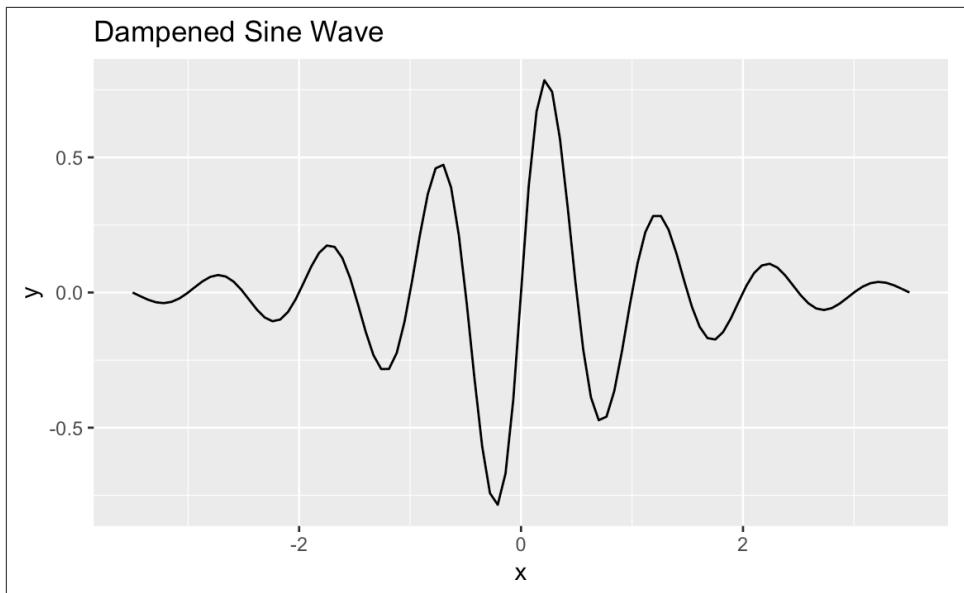


Figure 10-58. Damped sine wave plot

See Also

See [Recipe 15.3](#) for how to define a function.

10.25 Displaying Several Figures on One Page

Problem

You want to display several plots side by side on one page.

Solution

There are a number of ways to put `ggplot` graphics into a grid, but one of the easiest to use and understand is `patchwork` by Thomas Lin Pedersen. `patchwork` is not currently available on CRAN, but you can install it from GitHub using the `devtools` package:

```
devtools::install_github("thomasp85/patchwork")
```

After installing the package, you can use it to plot multiple `ggplot` objects using a `+` between the objects, then a call to `plot_layout` to arrange the images into a grid, as shown in [Figure 10-59](#). The example code here has four `ggplot` objects:

```
library(patchwork)
p1 + p2 + p3 + p4
```

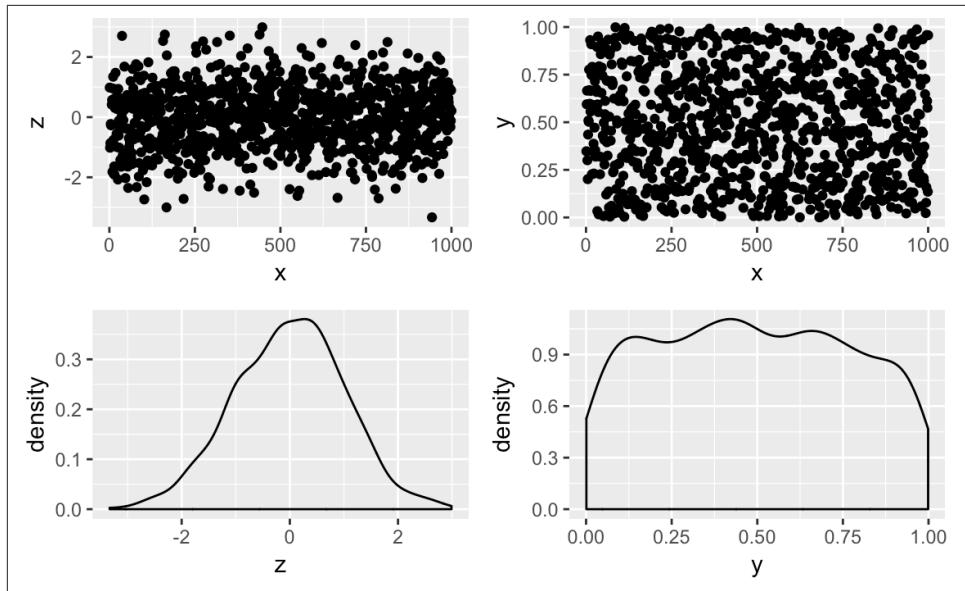


Figure 10-59. A patchwork plot

patchwork supports grouping with parentheses and using / to put groupings under other elements, as illustrated in [Figure 10-60](#):

`p3 / (p1 + p2 + p4)`

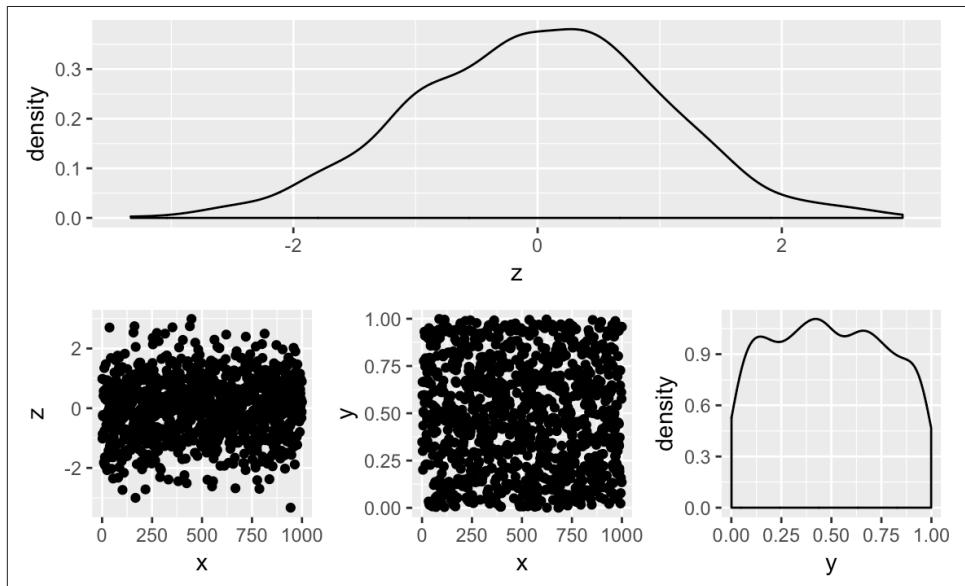


Figure 10-60. A patchwork 1 / 2 plot

Discussion

Let's use a multifigure plot to display four different beta distributions. Using `ggplot` and the `patchwork` package, we can create a 2×2 layout effect by creating four graphics objects and then printing them using the `+` notation from `patchwork`:

```
library(patchwork)

df <- data.frame(x = c(0, 1))

g1 <- ggplot(df) +
  aes(x) +
  stat_function(
    fun = function(x)
      dbeta(x, 2, 4)
  ) +
  ggtitle("First")

g2 <- ggplot(df) +
  aes(x) +
  stat_function(
    fun = function(x)
      dbeta(x, 4, 1)
  ) +
  ggtitle("Second")

g3 <- ggplot(df) +
  aes(x) +
  stat_function(
    fun = function(x)
      dbeta(x, 1, 1)
  ) +
  ggtitle("Third")

g4 <- ggplot(df) +
  aes(x) +
  stat_function(
    fun = function(x)
      dbeta(x, .5, .5)
  ) +
  ggtitle("Fourth")

g1 + g2 + g3 + g4 + plot_layout(ncol = 2, byrow = TRUE)
```

The output is shown in [Figure 10-61](#).

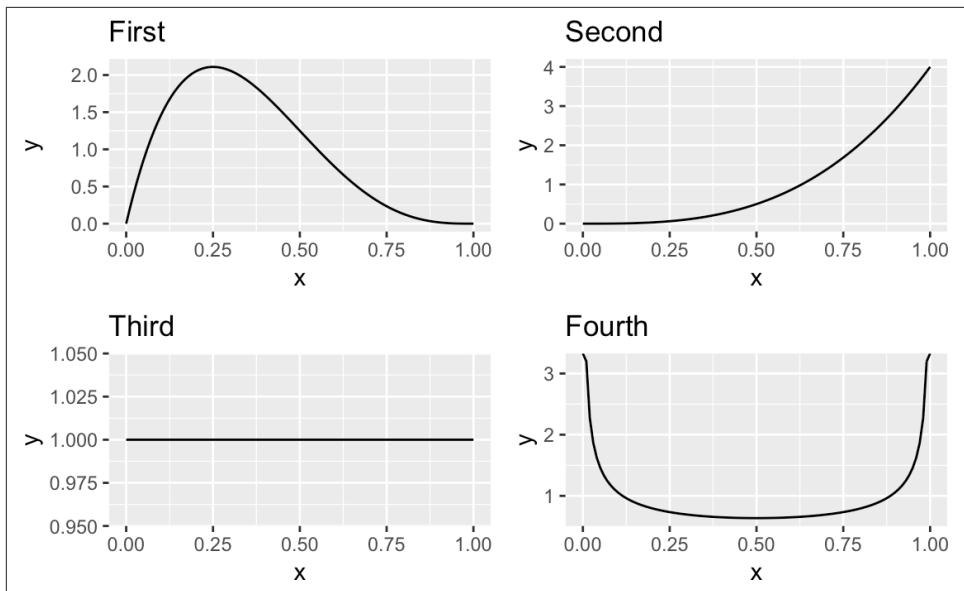


Figure 10-61. Four plots using patchwork

To lay the images out in column order, we could pass `byrow=FALSE` to `plot_layout`:

```
g1 + g2 + g3 + g4 + plot_layout(ncol = 2, byrow = FALSE)
```

See Also

[Recipe 8.11](#) discusses plotting density functions as we do here.

[Recipe 10.9](#) shows how you can create a matrix of plots using a `facet` function.

The `grid` package and the `lattice` package contain additional tools for multifigure layouts with base graphics.

10.26 Writing Your Plot to a File

Problem

You want to save your graphics in a file, such as a PNG, JPEG, or PostScript file.

Solution

With `ggplot` figures you can use `ggsave` to save a displayed image to a file. `ggsave` will make some default assumptions about size and file type for you, allowing you to specify only a filename:

```
ggsave("filename.jpg")
```

The file type is derived from the extension you use in the filename you pass to `ggsave`. You can control details of size, file type, and scale by passing parameters to `ggsave`. See `?ggsave` for specific details.

Discussion

In RStudio, a shortcut is to click on `Export` in the `Plots` window and then click on “Save as Image,” “Save as PDF,” or “Copy to Clipboard.” The save options will prompt you for a file type and a filename before writing the file. The “Copy to Clipboard” option can be handy if you are manually copying and pasting your graphics into a presentation or word processor.

Remember that the file will be written to your current working directory (unless you use an absolute filepath), so be certain you know which directory is your working directory before calling `savePlot`.

In a noninteractive script using `ggplot`, you can pass plot objects directly to `ggsave` so they need not be displayed before saving. In the prior recipe we created a plot object called `g1`. We can save it to a file like this:

```
ggsave("g1.png", plot = g1, units = "in", width = 5, height = 4)
```

Note that the units for `height` and `width` in `ggsave` are specified with the `units` parameter. In this case we used `in` for inches, but `ggsave` also supports `mm` and `cm` for the more metrically inclined.

See Also

See [Recipe 3.1](#) for more about the current working directory.

Linear Regression and ANOVA

In statistics, modeling is where we get down to business. Models quantify the relationships between our variables. Models let us make predictions.

A simple linear regression is the most basic model. It's just two variables and is modeled as a linear relationship with an error term:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

We are given the data for x and y . Our mission is to fit the model, which will give us the best estimates for β_0 and β_1 (see [Recipe 11.1](#)).

That generalizes naturally to multiple linear regression, where we have multiple variables on the righthand side of the relationship (see [Recipe 11.2](#)):

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \varepsilon_i$$

Statisticians call u , v , and w the *predictors* and y the *response*. Obviously, the model is useful only if there is a fairly linear relationship between the predictors and the response, but that requirement is much less restrictive than you might think. [Recipe 11.12](#) discusses transforming your variables into a (more) linear relationship so that you can use the well-developed machinery of linear regression.

The beauty of R is that anyone can build these linear models. The models are built by a function, `lm`, which returns a model object. From the model object, we get the coefficients (β_i) and regression statistics. It's easy. Really!

The horror of R is likewise that anyone can build these models. Nothing requires you to check that the model is reasonable, much less statistically significant. Before you blindly believe a model, check it! Most of the information you need is in the regression summary (see [Recipe 11.4](#)):

Is the model statistically significant?

Check the F statistic at the bottom of the summary.

Are the coefficients significant?

Check the coefficient's t statistics and p -values in the summary, or check their confidence intervals (see [Recipe 11.14](#)).

Is the model useful?

Check the R^2 near the bottom of the summary.

Does the model fit the data well?

Plot the residuals and check the regression diagnostics (see [Recipe 11.15](#) and [Recipe 11.16](#)).

Does the data satisfy the assumptions behind linear regression?

Check whether the diagnostics confirm that a linear model is reasonable for your data (see [Recipe 11.16](#)).

ANOVA

Analysis of variance (ANOVA) is a powerful statistical technique. First-year graduate students in statistics are taught ANOVA almost immediately because of its importance, both theoretical and practical. We are often amazed, however, at the extent to which people outside the field are unaware of its purpose and value.

Regression creates a model, and ANOVA is one method of evaluating such models. The mathematics of ANOVA are intertwined with the mathematics of regression, so statisticians usually present them together; we follow that tradition here.

ANOVA is actually a family of techniques that are connected by a common mathematical analysis. This chapter mentions several applications:

One-way ANOVA

This is the simplest application of ANOVA. Suppose you have data samples from several populations and are wondering whether the populations have different means. One-way ANOVA answers that question. If the populations have normal distributions, use the `oneway.test` function (see [Recipe 11.21](#)); otherwise, use the nonparametric version, the `kruskal.test` function (see [Recipe 11.24](#)).

Model comparison

When you add or delete a predictor variable in a linear regression, you want to know whether that change improved the model. The `anova` function compares two regression models and reports whether they are significantly different (see [Recipe 11.25](#)).

ANOVA table

The `anova` function can also construct the ANOVA table of a linear regression model, which includes the F statistic needed to gauge the model's statistical significance (see [Recipe 11.3](#)). This important table is discussed in nearly every textbook on regression.

Example Data

In many of the examples in this chapter, we start by creating example data using R's pseudorandom number generation capabilities. So at the beginning of each recipe, you may see something like the following:

```
set.seed(42)
x <- rnorm(100)
e <- rnorm(100, mean=0, sd=5)
y <- 5 + 15 * x + e
```

We use `set.seed` to set the random number generation seed so that if you run the example code on your machine you will get the same answer. In the preceding example, `x` is a vector of 100 draws from a standard normal (`mean=0, sd=1`) distribution. Then we create a little random noise called `e` from a normal distribution with `mean=0` and `sd= 5`. `y` is then calculated as `5 + 15 * x + e`. The idea behind creating example “toy” data rather than using “real-world” data is that with simulated data you can change the coefficients and parameters and see how the change impacts the resulting model. For example, you could increase the standard deviation of `e` in the example data and see what impact that has on the R^2 of your model.

See Also

There are many good texts on linear regression. One of our favorites is *Applied Linear Regression Models*, 4th ed., by Michael Kutner, Christopher Nachtsheim, and John Neter (McGraw-Hill/Irwin). We generally follow their terminology and conventions in this chapter.

We also like *Linear Models with R* by Julian Faraway (Chapman & Hall/CRC), because it illustrates regression using R and is quite readable. Earlier versions of Faraday's work are available free [online](#), too.

11.1 Performing Simple Linear Regression

Problem

You have two vectors, x and y , that hold paired observations: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. You believe there is a linear relationship between x and y , and you want to create a regression model of the relationship.

Solution

The `lm` function performs a linear regression and reports the coefficients.

If your data is in vectors:

```
lm(y ~ x)
```

Or if your data is in columns in a data frame:

```
lm(y ~ x, data = df)
```

Discussion

Simple linear regression involves two variables: a predictor (or independent) variable, often called x , and a response (or dependent) variable, often called y . The regression uses the *ordinary least-squares* (OLS) algorithm to fit the linear model:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

where β_0 and β_1 are the regression coefficients and ε_i are the error terms.

The `lm` function can perform linear regression. The main argument is a model formula, such as $y \sim x$. The formula has the response variable on the left of the tilde character (\sim) and the predictor variable on the right. The function estimates the regression coefficients, β_0 and β_1 , and reports them as the intercept and the coefficient of x , respectively:

```
set.seed(42)
x <- rnorm(100)
e <- rnorm(100, mean = 0, sd = 5)
y <- 5 + 15 * x + e

lm(y ~ x)
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Coefficients:
#> (Intercept)           x
#>       4.56          15.14
```

In this case, the regression equation is:

$$y_i = 4.56 + 15.14x_i + \varepsilon_i$$

It is quite common for data to be captured inside a data frame, in which case you want to perform a regression between two data frame columns. Here, x and y are columns of a data frame `df`:

```
df <- data.frame(x, y)
head(df)
#>   x     y
```

```
#> 1 1.371 31.57
#> 2 -0.565 1.75
#> 3 0.363 5.43
#> 4 0.633 23.74
#> 5 0.404 7.73
#> 6 -0.106 3.94
```

The `lm` function lets you specify a data frame by using the `data` parameter. If you do, the function will take the variables from the data frame and not from your workspace:

```
lm(y ~ x, data = df)           # Take x and y from df
#>
#> Call:
#> lm(formula = y ~ x, data = df)
#>
#> Coefficients:
#> (Intercept)          x
#>       4.56          15.14
```

11.2 Performing Multiple Linear Regression

Problem

You have several predictor variables (e.g., u , v , and w) and a response variable, y . You believe there is a linear relationship between the predictors and the response, and you want to perform a linear regression on the data.

Solution

Use the `lm` function. Specify the multiple predictors on the righthand side of the formula, separated by plus signs (+):

```
lm(y ~ u + v + w)
```

Discussion

Multiple linear regression is the obvious generalization of simple linear regression. It allows multiple predictor variables instead of one predictor variable and still uses OLS to compute the coefficients of a linear equation. The three-variable regression just given corresponds to this linear model:

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \varepsilon_i$$

R uses the `lm` function for both simple and multiple linear regression. You simply add more variables to the righthand side of the model formula. The output then shows the coefficients of the fitted model. Let's set up some example random normal data using the `rnorm` function:

```

set.seed(42)
u <- rnorm(100)
v <- rnorm(100, mean = 3, sd = 2)
w <- rnorm(100, mean = -3, sd = 1)
e <- rnorm(100, mean = 0, sd = 3)

```

Then we can create an equation using known coefficients to calculate our y variable:

```
y <- 5 + 4 * u + 3 * v + 2 * w + e
```

Now if we run a linear regression, we can see that R solves for the coefficients and gets pretty close to the actual values just used:

```

lm(y ~ u + v + w)
#>
#> Call:
#> lm(formula = y ~ u + v + w)
#>
#> Coefficients:
#> (Intercept)          u            v            w
#>       4.77        4.17       3.01       1.91

```

The `data` parameter of `lm` is especially valuable when the number of variables increases, since it's much easier to keep your data in one data frame than in many separate variables. Suppose your data is captured in a data frame, such as the `df` variable shown here:

```

df <- data.frame(y, u, v, w)
head(df)
#>      y      u      v      w
#> 1 16.67 1.371 5.402 -5.00
#> 2 14.96 -0.565 5.090 -2.67
#> 3 5.89  0.363 0.994 -1.83
#> 4 27.95  0.633 6.697 -0.94
#> 5 2.42   0.404 1.666 -4.38
#> 6 5.73  -0.106 3.211 -4.15

```

When you supply `df` to the `data` parameter of `lm`, R looks for the regression variables in the columns of the data frame:

```

lm(y ~ u + v + w, data = df)
#>
#> Call:
#> lm(formula = y ~ u + v + w, data = df)
#>
#> Coefficients:
#> (Intercept)          u            v            w
#>       4.77        4.17       3.01       1.91

```

See Also

See [Recipe 11.1](#) for simple linear regression.

11.3 Getting Regression Statistics

Problem

You want the critical statistics and information regarding your regression, such as R^2 , the F statistic, confidence intervals for the coefficients, residuals, the ANOVA table, and so forth.

Solution

Save the regression model in a variable, say `m`:

```
m <- lm(y ~ u + v + w)
```

Then use functions to extract regression statistics and information from the model:

`anova(m)`

ANOVA table

`coefficients(m)`

Model coefficients

`coef(m)`

Same as `coefficients(m)`

`confint(m)`

Confidence intervals for the regression coefficients

`deviance(m)`

Residual sum of squares

`effects(m)`

Vector of orthogonal effects

`fitted(m)`

Vector of fitted y values

`residuals(m)`

Model residuals

`resid(m)`

Same as `residuals(m)`

`summary(m)`

Key statistics, such as R^2 , the F statistic, and the residual standard error (σ)

`vcov(m)`

Variance-covariance matrix of the main parameters

Discussion

When we started using R, the documentation said to use the `lm` function to perform linear regression. So we did something like this, getting the output shown in [Recipe 11.2](#):

```
lm(y ~ u + v + w)
#>
#> Call:
#> lm(formula = y ~ u + v + w)
#>
#> Coefficients:
#> (Intercept)      u          v          w
#>        4.77       4.17       3.01       1.91
```

How disappointing! The output was nothing compared to other statistics packages such as SAS. Where is R^2 ? Where are the confidence intervals for the coefficients? Where is the F statistic, its p -value, and the ANOVA table?

Of course, all that information is available—you just have to ask for it. Other statistics systems dump everything and let you wade through it. R is more minimalist. It prints a bare-bones output and lets you request what more you want.

The `lm` function returns a model object that you can assign to a variable:

```
m <- lm(y ~ u + v + w)
```

From the model object, you can extract important information using specialized functions. The most important function is `summary`:

```
summary(m)
#>
#> Call:
#> lm(formula = y ~ u + v + w)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -5.383 -1.760 -0.312  1.856  6.984
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  4.770     0.969   4.92  3.5e-06 ***
#> u            4.173     0.260  16.07 < 2e-16 ***
#> v            3.013     0.148  20.31 < 2e-16 ***
#> w            1.905     0.266   7.15  1.7e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
#>
#> Residual standard error: 2.66 on 96 degrees of freedom
#> Multiple R-squared:  0.885, Adjusted R-squared:  0.882
#> F-statistic: 247 on 3 and 96 DF, p-value: <2e-16
```

The summary shows the estimated coefficients, the critical statistics (such as R^2 and the F statistic), and an estimate of σ , the standard error of the residuals. The summary is so important that there is an entire recipe devoted to understanding it ([Recipe 11.4](#)).

There are specialized extractor functions for other important information:

Model coefficients (point estimates)

```
coef(m)
#> (Intercept)      u          v          w
#>        4.77     4.17     3.01     1.91
```

Confidence intervals for model coefficients

```
confint(m)
#>              2.5 % 97.5 %
#> (Intercept) 2.85   6.69
#> u            3.66   4.69
#> v            2.72   3.31
#> w            1.38   2.43
```

Model residuals

```
resid(m)
#>    1     2     3     4     5     6     7     8     9
#> -0.5675 2.2880 0.0972 2.1474 -0.7169 -0.3617 1.0350 2.8040 -4.2496
#>    10    11    12    13    14    15    16    17    18
#> -0.2048 -0.6467 -2.5772 -2.9339 -1.9330 1.7800 -1.4400 -2.3989 0.9245
#>    19    20    21    22    23    24    25    26    27
#> -3.3663 2.6890 -1.4190 0.7871 0.0355 -0.3806 5.0459 -2.5011 3.4516
#>    28    29    30    31    32    33    34    35    36
#> 0.3371 -2.7099 -0.0761 2.0261 -1.3902 -2.7041 0.3953 2.7201 -0.0254
#>    37    38    39    40    41    42    43    44    45
#> -3.9887 -3.9011 -1.9458 -1.7701 -0.2614 2.0977 -1.3986 -3.1910 1.8439
#>    46    47    48    49    50    51    52    53    54
#> 0.8218 3.6273 -5.3832 0.2905 3.7878 1.9194 -2.4106 1.6855 -2.7964
#>    55    56    57    58    59    60    61    62    63
#> -1.3348 3.3549 -1.1525 2.4012 -0.5320 -4.9434 -2.4899 -3.2718 -1.6161
#>    64    65    66    67    68    69    70    71    72
#> -1.5119 -0.4493 -0.9869 5.6273 -4.4626 -1.7568 0.8099 5.0320 0.1689
#>    73    74    75    76    77    78    79    80    81
#> 3.5761 -4.8668 4.2781 -2.1386 -0.9739 -3.6380 0.5788 5.5664 6.9840
#>    82    83    84    85    86    87    88    89    90
#> -3.5119 1.2842 4.1445 -0.4630 -0.7867 -0.7565 1.6384 3.7578 1.8942
#>    91    92    93    94    95    96    97    98    99
#> 0.5542 -0.8662 1.2041 -1.7401 -0.7261 3.2701 1.4012 0.9476 -0.9140
#>    100
#> 2.4278
```

Residual sum of squares

```
deviance(m)
#> [1] 679
```

ANOVA table

```
anova(m)
#> Analysis of Variance Table
#>
#> Response: y
#>           Df Sum Sq Mean Sq F value    Pr(>F)
#> u          1 1776   1776  251.0 < 2e-16 ***
#> v          1 3097   3097  437.7 < 2e-16 ***
#> w          1  362    362   51.1 1.7e-10 ***
#> Residuals  96  679      7
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If you find it annoying to save the model in a variable, you are welcome to use one-liners such as this:

```
summary(lm(y ~ u + v + w))
```

Or you can use `magrittr` pipes:

```
lm(y ~ u + v + w) %>%
  summary
```

See Also

See [Recipe 11.4](#) for more on the regression summary. See [Recipe 11.17](#) for regression statistics specific to model diagnostics.

11.4 Understanding the Regression Summary

Problem

You created a linear regression model, `m`. However, you are confused by the output from `summary(m)`.

Discussion

The model summary is important because it links you to the most critical regression statistics. Here is the model summary from [Recipe 11.3](#):

```
summary(m)
#>
#> Call:
#> lm(formula = y ~ u + v + w)
#>
#> Residuals:
#>   Min     1Q Median     3Q    Max
#> -5.383 -1.760 -0.312  1.856  6.984
#>
#> Coefficients:
```

```

#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 4.770     0.969     4.92  3.5e-06 ***
#> u            4.173     0.260    16.07 < 2e-16 ***
#> v            3.013     0.148    20.31 < 2e-16 ***
#> w            1.905     0.266     7.15  1.7e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.66 on 96 degrees of freedom
#> Multiple R-squared:  0.885, Adjusted R-squared:  0.882
#> F-statistic: 247 on 3 and 96 DF, p-value: <2e-16

```

Let's dissect this summary by section. We'll read it from top to bottom, even though the most important statistic (the F statistic) appears at the end:

Call

```
#> lm(formula = y ~ u + v + w)
```

This shows how `lm` was called when it created the model, which is important for putting this summary into the proper context.

Residuals statistics

```

#> Residuals:
#>   Min    1Q  Median    3Q   Max
#> -5.383 -1.760 -0.312  1.856  6.984

```

Ideally, the regression residuals would have a perfect normal distribution. These statistics help you identify possible deviations from normality. The OLS algorithm is mathematically guaranteed to produce residuals with a mean of zero,¹ hence the sign of the median indicates the skew's direction and the magnitude of the median indicates the extent. In this case the median is negative, which suggests some skew to the left.

If the residuals have a nice bell-shaped distribution, then the first quartile (1Q) and third quartile (3Q) should have about the same magnitude. In this example, the larger magnitude of 3Q versus 1Q (1.856 versus 1.76) indicates a slight skew to the right in our data, although the negative median makes the situation less clear-cut.

The `Min` and `Max` residuals offer a quick way to detect extreme outliers in the data, since extreme outliers (in the response variable) produce large residuals.

Coefficients

```

#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 4.770     0.969     4.92  3.5e-06 ***
#> u            4.173     0.260    16.07 < 2e-16 ***

```

¹ Unless you performed the linear regression without an intercept term (see [Recipe 11.5](#)).

```
#> v           3.013      0.148    20.31 < 2e-16 ***
#> w           1.905      0.266     7.15  1.7e-10 ***
```

The column labeled `Estimate` contains the estimated regression coefficients as calculated by ordinary least squares.

Theoretically, if a variable's coefficient is zero then the variable is worthless; it adds nothing to the model. Yet the coefficients shown here are only estimates, and they will never be exactly zero. We therefore ask: statistically speaking, how likely is it that the true coefficient is zero? That is the purpose of the *t* statistics and the *p*-values, which in the summary are labeled (respectively) `t value` and `Pr(>|t|)`.

The *p*-value is a probability. It gauges the likelihood that the coefficient is *not* significant, so smaller is better. Big is bad because it indicates a high likelihood of insignificance. In this example, the *p*-value for the `u` coefficient is a mere 0.00106, so `u` is likely significant. The *p*-value for `w`, however, is 0.05744; this is just over our conventional limit of 0.05, which suggests that `w` is likely insignificant.² Variables with large *p*-values are candidates for elimination.

A handy feature is that R flags the significant variables for quick identification. Did you notice the extreme righthand column containing triple asterisks (**)? Other values you might see in this column are double asterisks (**), a single asterisk (*), and a period (.). This column highlights the significant variables. The line labeled `Signif. codes` at the bottom of the `Coefficients` section gives a cryptic guide to the flags' meanings. You can interpret them as follows:

Significance indication	Meaning
***	<i>p</i> -value between 0 and 0.001
**	<i>p</i> -value between 0.001 and 0.01
*	<i>p</i> -value between 0.01 and 0.05
.	<i>p</i> -value between 0.05 and 0.1
(blank)	<i>p</i> -value between 0.1 and 1.0

The column labeled `Std. Error` is the standard error of the estimated coefficient. The column labeled `t value` is the *t* statistic from which the *p*-value was calculated.

² The significance level of $\alpha = 0.05$ is the convention observed in this book. Your application might instead use $\alpha = 0.10$, $\alpha = 0.01$, or some other value. See the introduction to [Chapter 9](#).

Residual standard error

```
# Residual standard error: 2.66 on 96 degrees of freedom
```

This reports the standard error of the residuals (σ)—that is, the sample standard deviation of ε .

R² (coefficient of determination)

```
# Multiple R-squared:  0.885,   Adjusted R-squared:  0.882
```

R^2 is a measure of the model's quality. Bigger is better. Mathematically, it is the fraction of the variance of y that is explained by the regression model. The remaining variance is not explained by the model, so it must be due to other factors (i.e., unknown variables or sampling variability). In this case, the model explains 0.885 (88.5%) of the variance of y , and the remaining 0.115 (11.5%) is unexplained.

That being said, we strongly suggest using the adjusted rather than the basic R^2 . The adjusted value accounts for the number of variables in your model and so is a more realistic assessment of its effectiveness. In this case, then, we would use 0.882, not 0.885.

F statistic

```
# F-statistic: 246.6 on 3 and 96 DF,  p-value: < 2.2e-16
```

The F statistic tells you whether the model is significant or insignificant. The model is significant if any of the coefficients are nonzero (i.e., if $\beta_i \neq 0$ for some i). It is insignificant if all coefficients are zero ($\beta_1 = \beta_2 = \dots = \beta_n = 0$).

Conventionally, a p -value of less than 0.05 indicates that the model is likely significant (one or more β_i are nonzero), whereas values exceeding 0.05 indicate that the model is likely not significant. Here, the probability is only 2.2e-16 that our model is insignificant. That's good.

Most people look at the R^2 statistic first. The statistician wisely starts with the F statistic, because if the model is not significant then nothing else matters.

See Also

See [Recipe 11.3](#) for more on extracting statistics and information from the model object.

11.5 Performing Linear Regression Without an Intercept

Problem

You want to perform a linear regression, but you want to force the intercept to be zero.

Solution

Add "+ 0" to the righthand side of your regression formula. That will force `lm` to fit the model with a zero intercept:

```
lm(y ~ x + 0)
```

The corresponding regression equation is:

$$y_i = \beta x_i + \varepsilon_i$$

Discussion

Linear regression ordinarily includes an intercept term, so that is the default in R. In rare cases, however, you may want to fit the data while assuming that the intercept is zero. In this case you make a modeling assumption: when x is zero, y should be zero.

When you force a zero intercept, the `lm` output includes a coefficient for x but no intercept for y , as shown here:

```
lm(y ~ x + 0)
#>
#> Call:
#> lm(formula = y ~ x + 0)
#>
#> Coefficients:
#>   x
#> 4.3
```

We strongly suggest you check that modeling assumption before proceeding. Perform a regression with an intercept; then see if the intercept could plausibly be zero. Check the intercept's confidence interval. In this example, the confidence interval is (6.26, 8.84):

```
confint(lm(y ~ x))
#>           2.5 % 97.5 %
#> (Intercept) 6.26  8.84
#> x           2.82  5.31
```

Because the confidence interval does not contain zero, it is *not* statistically plausible that the intercept could be zero. So in this case, it is not reasonable to rerun the regression while forcing a zero intercept.

11.6 Regressing Only Variables That Highly Correlate with Your Dependent Variable

Problem

You have a data frame with many variables and you want to build a multiple linear regression using only the variables that are highly correlated to your response (dependent) variable.

Solution

If `df` is our data frame containing both our response (dependent) and all our predictor (independent) variables and `dep_var` is our response variable, we can figure out our best predictors and then use them in a linear regression. If we want the top four predictor variables, we can use this:

```
best_pred <- df %>%
  select(-dep_var) %>%
  map_dbl(cor, y = df$dep_var) %>%
  sort(decreasing = TRUE) %>%
  .[1:4] %>%
  names %>%
  df[.]  
  
mod <- lm(df$dep_var ~ as.matrix(best_pred))
```

This recipe is a combination of many different pieces of logic used elsewhere in this book. We will describe each step here, and then walk through it in the Discussion using some example data.

First we drop the response variable out of our pipe chain so that we have only our predictor variables in our data flow:

```
df %>%
  select(-dep_var)
```

Then we use `map_dbl` from `purrr` to perform a pairwise correlation on each column relative to the response variable:

```
map_dbl(cor, y = df$dep_var) %>%
```

We then take the resulting correlations and sort them in decreasing order:

```
sort(decreasing = TRUE) %>%
```

We want only the top four correlated variables, so we select the top four records in the resulting vector:

```
.[1:4] %>%
```

And we don't need the correlation values, only the names of the rows—which are the variable names from our original data frame, `df`:

```
names %>%
```

Then we can pass those names into our subsetting brackets to select only the columns with names matching the ones we want:

```
df[.]
```

Our pipe chain assigns the resulting data frame into `best_pred`. We can then use `best_pred` as the predictor variables in our regression and we can use `df$dep_var` as the response:

```
mod <- lm(df$dep_var ~ as.matrix(best_pred))
```

Discussion

By combining the mapping functions discussed in [Recipe 6.4](#), we can create a recipe to remove low-correlation variables from a set of predictors and use the high-correlation predictors in a regression.

We have an example data frame that contains six predictor variables named `pred1` through `pred6`. The response variable is named `resp`. Let's walk that data frame through our logic and see how it works.

Loading the data and dropping the `resp` variable is pretty straightforward, so let's look at the result of mapping the `cor` function:

```
# loads the pred data frame
load("./data/pred.rdata")

pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp)
#> pred1 pred2 pred3 pred4 pred5 pred6
#> 0.573 0.279 0.753 0.799 0.322 0.607
```

The output is a named vector of values where the names are the variable names and the values are the pairwise correlations between each predictor variable and `resp`, the response variable.

If we sort this vector, we get the correlations in decreasing order:

```
pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp) %>%
  sort(decreasing = TRUE)
#> pred4 pred3 pred6 pred1 pred2
#> 0.799 0.753 0.607 0.573 0.322 0.279
```

Using subsetting allows us to select the top four records. The `.` operator is a special operator that tells the pipe where to put the result of the prior step:

```
pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp) %>%
  sort(decreasing = TRUE) %>%
  .[1:4]
#> pred4 pred3 pred6 pred1
#> 0.799 0.753 0.607 0.573
```

We then use the `names` function to extract the names from our vector. The names are the names of the columns we ultimately want to use as our independent variables:

```
pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp) %>%
  sort(decreasing = TRUE) %>%
  .[1:4] %>%
  names
#> [1] "pred4" "pred3" "pred6" "pred1"
```

When we pass the vector of names into `pred[.]`, the names are used to select columns from the `pred` data frame. We then use `head` to select only the top six rows for easier illustration:

```
pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp) %>%
  sort(decreasing = TRUE) %>%
  .[1:4] %>%
  names %>%
  pred[.] %>%
  head
#>   pred4   pred3   pred6   pred1
#> 1  7.252  1.5127  0.560  0.206
#> 2  2.076  0.2579 -0.124 -0.361
#> 3 -0.649  0.0884  0.657  0.758
#> 4  1.365 -0.1209  0.122 -0.727
#> 5 -5.444 -1.1943 -0.391 -1.368
#> 6  2.554  0.6120  1.273  0.433
```

Now let's bring it all together and pass the resulting data into the regression:

```
best_pred <- pred %>%
  select(-resp) %>%
  map_dbl(cor, y = pred$resp) %>%
  sort(decreasing = TRUE) %>%
  .[1:4] %>%
  names %>%
  pred[.]

mod <- lm(pred$resp ~ as.matrix(best_pred))
```

```

summary(mod)
#>
#> Call:
#> lm(formula = pred$resp ~ as.matrix(best_pred))
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -1.485 -0.619  0.189  0.562  1.398
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)    
#> (Intercept)   1.117     0.340    3.28   0.0051 **  
#> as.matrix(best_pred)pred4   0.523     0.207    2.53   0.0231 *   
#> as.matrix(best_pred)pred3  -0.693     0.870   -0.80   0.4382    
#> as.matrix(best_pred)pred6   1.160     0.682    1.70   0.1095    
#> as.matrix(best_pred)pred1   0.343     0.359    0.95   0.3549    
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.927 on 15 degrees of freedom
#> Multiple R-squared:  0.838, Adjusted R-squared:  0.795 
#> F-statistic: 19.4 on 4 and 15 DF,  p-value: 8.59e-06

```

11.7 Performing Linear Regression with Interaction Terms

Problem

You want to include an interaction term in your regression.

Solution

The R syntax for regression formulas lets you specify interaction terms. To indicate the interaction of two variables, u and v , we separate their names with an asterisk (*):

```
lm(y ~ u * v)
```

This corresponds to the model $y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 u_i v_i + \varepsilon_i$, which includes the first-order interaction term $\beta_3 u_i v_i$.

Discussion

In regression, an interaction occurs when the product of two predictor variables is also a significant predictor (i.e., in addition to the predictor variables themselves). Suppose we have two predictors, u and v , and want to include their interaction in the regression. This is expressed by the following equation:

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 u_i v_i + \varepsilon_i$$

Here the product term, $\beta_3 u_i v_i$, is called the *interaction term*. The R formula for that equation is:

`y ~ u * v`

When you write `y ~ u * v`, R automatically includes u , v , and their product in the model. This is for a good reason. If a model includes an interaction term, such as $\beta_3 u_i v_i$, then regression theory tells us the model should also contain the constituent variables u_i and v_i .

Likewise, if you have three predictors (u , v , and w) and want to include all their interactions, separate them by asterisks:

`y ~ u * v * w`

This corresponds to the regression equation:

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \beta_4 u_i v_i + \beta_5 u_i w_i + \beta_6 v_i w_i + \beta_7 u_i v_i w_i + \varepsilon_i$$

Now we have all the first-order interactions and a second-order interaction ($\beta_7 u_i v_i w_i$).

Sometimes, however, you may not want every possible interaction. You can explicitly specify a single product by using the colon operator (`:`). For example, `u:v:w` denotes the product term $\beta u_i v_i w_i$ but without all possible interactions. So the R formula:

`y ~ u + v + w + u:v:w`

corresponds to the regression equation:

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \beta_4 u_i v_i w_i + \varepsilon_i$$

It might seem odd that a colon (`:`) means pure multiplication while an asterisk (`*`) means both multiplication and inclusion of constituent terms. Again, this is because we normally incorporate the constituents when we include their interaction, so making that approach the default for `*` makes sense.

There is some additional syntax for easily specifying many interactions:

`(u + v + ... + w)^2`

Include all variables (u, v, \dots, w) and all their first-order interactions.

`(u + v + ... + w)^3`

Include all variables, all their first-order interactions, and all their second-order interactions.

`(u + v + ... + w)^4`

And so forth.

Both the asterisk (`*`) and the colon (`:`) follow a “distributive law,” so the following notations are also allowed:

```
x*(u + v + ... + w)
```

Same as $x^*u + x^*v + \dots + x^*w$ (which is the same as $x + u + v + \dots + w + x:u + x:v + \dots + x:w$)

```
x:(u + v + ... + w)
```

Same as $x:u + x:v + \dots + x:w$

All this syntax gives you some flexibility in writing your formula. For example, these three formulas are equivalent:

```
y ~ u * v
```

```
y ~ u + v + u:v
```

```
y ~ (u + v) ^ 2
```

They all define the same regression equation, $y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 u_i v_i + \varepsilon_i$.

See Also

The full syntax for formulas is richer than described here. See *R in a Nutshell* or the [R Language Definition](#) for more details.

11.8 Selecting the Best Regression Variables

Problem

You are creating a new regression model or improving an existing model. You have the luxury of many regression variables, and you want to select the best subset of those variables.

Solution

The `step` function can perform stepwise regression, either forward or backward. Backward stepwise regression starts with many variables and removes the underperformers:

```
full.model <- lm(y ~ x1 + x2 + x3 + x4)
reduced.model <- step(full.model, direction = "backward")
```

Forward stepwise regression starts with a few variables and adds new ones to improve the model until it cannot be improved further:

```
min.model <- lm(y ~ 1)
fwd.model <-
  step(min.model,
    direction = "forward",
    scope = (~ x1 + x2 + x3 + x4))
```

Discussion

When you have many predictors, it can be quite difficult to choose the best subset. Adding and removing individual variables affects the overall mix, so the search for “the best” can become tedious.

The `step` function automates that search. Backward stepwise regression is the easiest approach. Start with a model that includes all the predictors. We call that the *full model*. The model summary, shown here, indicates that not all predictors are statistically significant:

```
# example data
set.seed(4)
n <- 150
x1 <- rnorm(n)
x2 <- rnorm(n, 1, 2)
x3 <- rnorm(n, 3, 1)
x4 <- rnorm(n,-2, 2)
e <- rnorm(n, 0, 3)
y <- 4 + x1 + 5 * x3 + e

# build the model
full.model <- lm(y ~ x1 + x2 + x3 + x4)
summary(full.model)
#>
#> Call:
#> lm(formula = y ~ x1 + x2 + x3 + x4)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -8.032 -1.774  0.158  2.032  6.626
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 3.40224   0.80767   4.21  4.4e-05 ***
#> x1          0.53937   0.25935   2.08   0.039 *
#> x2          0.16831   0.12291   1.37   0.173
#> x3          5.17410   0.23983  21.57 < 2e-16 ***
#> x4         -0.00982   0.12954  -0.08   0.940
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.92 on 145 degrees of freedom
#> Multiple R-squared:  0.77,  Adjusted R-squared:  0.763
#> F-statistic: 121 on 4 and 145 DF,  p-value: <2e-16
```

We want to eliminate the insignificant variables, so we use `step` to incrementally eliminate the underperformers. The result is called the *reduced model*:

```
reduced.model <- step(full.model, direction="backward")
#> Start:  AIC=327
#> y ~ x1 + x2 + x3 + x4
```

```

#>
#>      Df Sum of Sq  RSS AIC
#> - x4     1          0 1240 325
#> - x2     1         16 1256 327
#> <none>           1240 327
#> - x1     1         37 1277 329
#> - x3     1        3979 5219 540
#>
#> Step:  AIC=325
#> y ~ x1 + x2 + x3
#>
#>      Df Sum of Sq  RSS AIC
#> - x2     1         16 1256 325
#> <none>           1240 325
#> - x1     1         37 1277 327
#> - x3     1        3988 5228 539
#>
#> Step:  AIC=325
#> y ~ x1 + x3
#>
#>      Df Sum of Sq  RSS AIC
#> <none>           1256 325
#> - x1     1         44 1300 328
#> - x3     1        3974 5230 537

```

The output from `step` shows the sequence of models that it explored. In this case, `step` removed `x2` and `x4` and left only `x1` and `x3` in the final (reduced) model. The summary of the reduced model shows that it contains only significant predictors:

```

summary(reduced.model)
#>
#> Call:
#> lm(formula = y ~ x1 + x3)
#>
#> Residuals:
#>   Min     1Q Median     3Q    Max
#> -8.148 -1.850 -0.055  2.026  6.550
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  3.648     0.751   4.86   3e-06 ***
#> x1          0.582     0.255   2.28   0.024 *
#> x3          5.147     0.239  21.57 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
#>
#> Residual standard error: 2.92 on 147 degrees of freedom
#> Multiple R-squared:  0.767, Adjusted R-squared:  0.763
#> F-statistic: 241 on 2 and 147 DF, p-value: <2e-16

```

Backward stepwise regression is easy, but sometimes it's not feasible to start with "everything" because you have too many candidate variables. In that case use forward

stepwise regression, which will start with nothing and incrementally add variables that improve the regression. It stops when no further improvement is possible.

A model that “starts with nothing” may look odd at first:

```
min.model <- lm(y ~ 1)
```

This is a model with a response variable (y) but no predictor variables. (All the fitted values for y are simply the mean of y , which is what you would guess if no predictors were available.)

We must tell `step` which candidate variables are available for inclusion in the model. That is the purpose of the `scope` argument. `scope` is a formula with nothing on the lefthand side of the tilde (~) and candidate variables on the righthand side:

```
fwd.model <- step(  
  min.model,  
  direction = "forward",  
  scope = (~ x1 + x2 + x3 + x4),  
  trace = 0  
)
```

Here we see that x_1 , x_2 , x_3 , and x_4 are all candidates for inclusion. (We also included `trace = 0` to inhibit the voluminous output from `step`.) The resulting model has two significant predictors and no insignificant predictors:

```
summary(fwd.model)  
#>  
#> Call:  
#> lm(formula = y ~ x3 + x1)  
#>  
#> Residuals:  
#>   Min     1Q Median     3Q    Max  
#> -8.148 -1.850 -0.055  2.026  6.550  
#>  
#> Coefficients:  
#>             Estimate Std. Error t value Pr(>|t|)  
#> (Intercept)  3.648     0.751    4.86   3e-06 ***  
#> x3          5.147     0.239   21.57   <2e-16 ***  
#> x1          0.582     0.255    2.28    0.024 *  
#> ---  
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1  
#>  
#> Residual standard error: 2.92 on 147 degrees of freedom  
#> Multiple R-squared:  0.767, Adjusted R-squared:  0.763  
#> F-statistic: 241 on 2 and 147 DF, p-value: <2e-16
```

The step-forward algorithm reached the same model as the step-backward model by including x_1 and x_3 but excluding x_2 and x_4 . This is a toy example, so that is not surprising. In real applications, we suggest trying both the forward and backward regression and then comparing the results. You might be surprised.

Finally, don't get carried away with stepwise regression. It is not a panacea, it cannot turn junk into gold, and it is definitely not a substitute for choosing predictors carefully and wisely. You might think: "Oh boy! I can generate every possible interaction term for my model, then let `step` choose the best ones! What a model I'll get!" You'd be thinking of something like this, which starts with all possible interactions and then tries to reduce the model:

```
full.model <- lm(y ~ (x1 + x2 + x3 + x4) ^ 4)
reduced.model <- step(full.model, direction = "backward")
#> Start:  AIC=337
#> y ~ (x1 + x2 + x3 + x4)^4
#>
#>          Df Sum of Sq  RSS AIC
#> - x1:x2:x3:x4  1     0.0321 1145 335
#> <none>                 1145 337
#>
#> Step:  AIC=335
#> y ~ x1 + x2 + x3 + x4 + x1:x2 + x1:x3 + x1:x4 + x2:x3 + x2:x4 +
#>      x3:x4 + x1:x2:x3 + x1:x2:x4 + x1:x3:x4 + x2:x3:x4
#>
#>          Df Sum of Sq  RSS AIC
#> - x2:x3:x4  1      0.76 1146 333
#> - x1:x3:x4  1      8.37 1154 334
#> <none>                 1145 335
#> - x1:x2:x4  1     20.95 1166 336
#> - x1:x2:x3  1     25.18 1170 336
#>
#> Step:  AIC=333
#> y ~ x1 + x2 + x3 + x4 + x1:x2 + x1:x3 + x1:x4 + x2:x3 + x2:x4 +
#>      x3:x4 + x1:x2:x3 + x1:x2:x4 + x1:x3:x4
#>
#>          Df Sum of Sq  RSS AIC
#> - x1:x3:x4  1      8.74 1155 332
#> <none>                 1146 333
#> - x1:x2:x4  1     21.72 1168 334
#> - x1:x2:x3  1     26.51 1172 334
#>
#> Step:  AIC=332
#> y ~ x1 + x2 + x3 + x4 + x1:x2 + x1:x3 + x1:x4 + x2:x3 + x2:x4 +
#>      x3:x4 + x1:x2:x3 + x1:x2:x4
#>
#>          Df Sum of Sq  RSS AIC
#> - x3:x4     1      0.29 1155 330
#> <none>                 1155 332
#> - x1:x2:x4  1     23.24 1178 333
#> - x1:x2:x3  1     31.11 1186 334
#>
#> Step:  AIC=330
#> y ~ x1 + x2 + x3 + x4 + x1:x2 + x1:x3 + x1:x4 + x2:x3 + x2:x4 +
#>      x1:x2:x3 + x1:x2:x4
#>
```

```
#>           Df Sum of Sq  RSS AIC
#> <none>              1155 330
#> - x1:x2:x4  1      23.4 1178 331
#> - x1:x2:x3  1      31.5 1187 332
```

This does not work well. Most of the interaction terms are meaningless. The `step` function becomes overwhelmed, and you are left with many insignificant terms.

See Also

See [Recipe 11.25](#).

11.9 Regressing on a Subset of Your Data

Problem

You want to fit a linear model to a subset of your data, not to the entire dataset.

Solution

The `lm` function has a `subset` parameter that specifies which data elements should be used for fitting. The parameter's value can be any index expression that could index your data. This shows a fitting that uses only the first 100 observations:

```
lm(y ~ x1, subset=1:100)          # Use only x[1:100]
```

Discussion

You will often want to regress only a subset of your data. This can happen, for example, when you're using in-sample data to create the model and out-of-sample data to test it.

The `lm` function has a parameter, `subset`, that selects the observations used for fitting. The value of `subset` is a vector. It can be a vector of index values, in which case `lm` selects only the indicated observations from your data. It can also be a logical vector, the same length as your data, in which case `lm` selects the observations with a corresponding `TRUE`.

Suppose you have 1,000 observations of (x, y) pairs and want to fit your model using only the first half of those observations. Use a `subset` parameter of `1:500`, indicating `lm` should use observations 1 through 500:

```
## example data
n <- 1000
x <- rnorm(n)
e <- rnorm(n, 0, .5)
y <- 3 + 2 * x + e
lm(y ~ x, subset = 1:500)
```

```
#>
#> Call:
#> lm(formula = y ~ x, subset = 1:500)
#>
#> Coefficients:
#> (Intercept)      x
#>       3          2
```

More generally, you can use the expression `1:floor(length(x)/2)` to select the first half of your data, regardless of size:

```
lm(y ~ x, subset = 1:floor(length(x) / 2))
#>
#> Call:
#> lm(formula = y ~ x, subset = 1:floor(length(x)/2))
#>
#> Coefficients:
#> (Intercept)      x
#>       3          2
```

Let's say your data was collected in several labs and you have a factor, `lab`, that identifies the lab of origin. You can limit your regression to observations collected in New Jersey by using a logical vector that is TRUE only for those observations:

```
load('./data/lab_df.rdata')
lm(y ~ x, subset = (lab == "NJ"), data = lab_df)
#>
#> Call:
#> lm(formula = y ~ x, data = lab_df, subset = (lab == "NJ"))
#>
#> Coefficients:
#> (Intercept)      x
#>       2.58       5.03
```

11.10 Using an Expression Inside a Regression Formula

Problem

You want to regress on calculated values, not simple variables, but the syntax of a regression formula seems to forbid that.

Solution

Embed the expressions for the calculated values inside the `I(...)` operator. That will force R to calculate the expression and use the calculated value for the regression.

Discussion

If you want to regress on the sum of u and v , then this is your regression equation:

$$y_i = \beta_0 + \beta_1(u_i + v_i) + \varepsilon_i$$

How do you write that equation as a regression formula? This won't work:

```
lm(y ~ u + v) # Not quite right
```

Here R will interpret `u` and `v` as two separate predictors, each with its own regression coefficient. Likewise, suppose your regression equation is:

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 u_i^2 + \varepsilon_i$$

This won't work:

```
lm(y ~ u + u ^ 2) # That's an interaction, not a quadratic term
```

R will interpret `u^2` as an interaction term (see [Recipe 11.7](#)) and not as the square of `u`.

The solution is to surround the expressions by the `I(...)` operator, which inhibits an expression from being interpreted as a regression formula. Instead, it forces R to calculate the expression's value and then incorporate that value directly into the regression. Thus, the first example becomes:

```
lm(y ~ I(u + v))
```

In response to that command, R computes `u + v` and then regresses `y` on the sum.

For the second example we use:

```
lm(y ~ u + I(u ^ 2))
```

Here R computes the square of `u` and then regresses on the sum `u + u ^ 2`.



All the basic binary operators (`+`, `-`, `*`, `/`, `^`) have special meanings inside a regression formula. For this reason, you must use the `I(...)` operator whenever you incorporate calculated values into a regression.

A beautiful aspect of these embedded transformations is that R remembers them and applies them when you make predictions from the model. Consider the quadratic model described by the second example. It uses `u` and `u^2`, but we supply the value of `u` only and R does the heavy lifting. We don't need to calculate the square of `u` ourselves:

```
load('./data/df_squared.rdata')
m <- lm(y ~ u + I(u ^ 2), data = df_squared)
predict(m, newdata = data.frame(u = 13.4))
#>    1
#> 877
```

See Also

See [Recipe 11.11](#) for the special case of regression on a polynomial. See [Recipe 11.12](#) for incorporating other data transformations into the regression.

11.11 Regressing on a Polynomial

Problem

You want to regress y on a polynomial of x .

Solution

Use the `poly(x, n)` function in your regression formula to regress on an n -degree polynomial of x . This example models y as a cubic function of x :

```
lm(y ~ poly(x, 3, raw = TRUE))
```

The example's formula corresponds to the following cubic regression equation:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \varepsilon_i$$

Discussion

When people first use a polynomial model in R, they often do something clunky like this:

```
x_sq <- x ^ 2
x_cub <- x ^ 3
m <- lm(y ~ x + x_sq + x_cub)
```

Obviously, this is quite annoying, and it litters their workspace with extra variables.

It's much easier to write:

```
m <- lm(y ~ poly(x, 3, raw = TRUE))
```

The `raw = TRUE` is necessary. Without it, the `poly` function computes orthogonal polynomials instead of simple polynomials.

Beyond the convenience, a huge advantage is that R will calculate all those powers of x when you make predictions from the model (see [Recipe 11.19](#)). Without that, you are stuck calculating x^2 and x^3 yourself every time you employ the model.

Here is another good reason to use `poly`. You cannot write your regression formula in this way:

```
lm(y ~ x + x^2 + x^3)      # Does not do what you think!
```

R will interpret x^2 and x^3 as interaction terms, not as powers of x . The resulting model is a one-term linear regression, completely unlike your expectation. You could write the regression formula like this:

```
lm(y ~ x + I(x ^ 2) + I(x ^ 3))
```

But that's getting pretty verbose. Just use `poly`.

See Also

See [Recipe 11.7](#) for more about interaction terms. See [Recipe 11.12](#) for other transformations on regression data.

11.12 Regressing on Transformed Data

Problem

You want to build a regression model for x and y , but they do not have a linear relationship.

Solution

You can embed the needed transformation inside the regression formula. If, for example, y must be transformed into $\log(y)$, then the regression formula becomes:

```
lm(log(y) ~ x)
```

Discussion

A critical assumption behind the `lm` function for regression is that the variables have a linear relationship. To the extent this assumption is false, the resulting regression becomes meaningless.

Fortunately, many datasets can be transformed into a linear relationship before applying `lm`.

[Figure 11-1](#) shows an example of exponential decay. The left panel shows the original data, z . The dotted line shows a linear regression on the original data; clearly, it's a lousy fit.

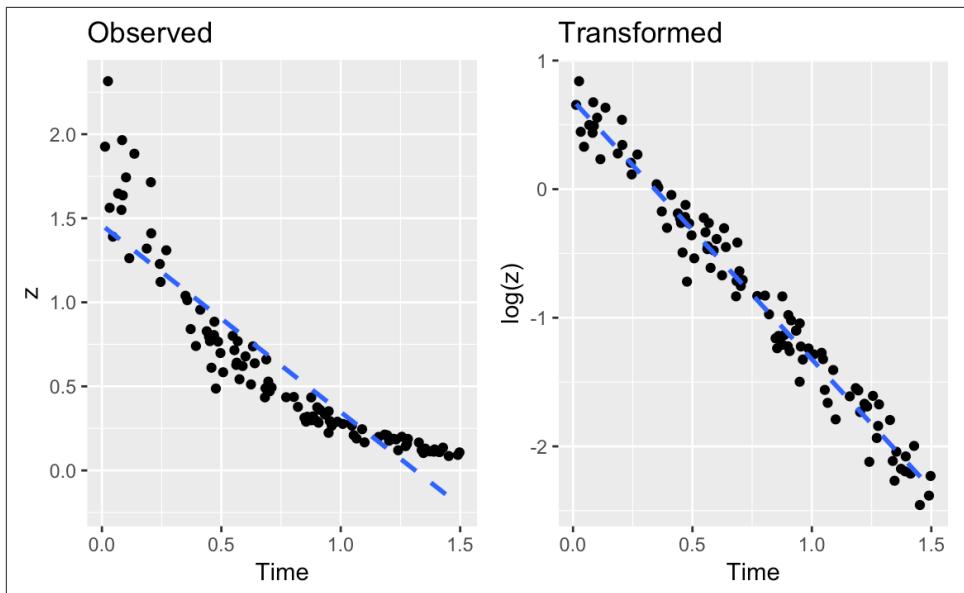


Figure 11-1. Example of a data transform

If the data is really exponential, then a possible model is:

$$z = \exp[\beta_0 + \beta_1 t + \varepsilon]$$

where t is time and $\exp[]$ is the exponential function (e^x). This is not linear, of course, but we can linearize it by taking logarithms:

$$\log(z) = \beta_0 + \beta_1 t + \varepsilon$$

In R, that regression is simple because we can embed the log transform directly into the regression formula:

```
# read in our example data
load(file = './data/df_decay.rdata')
z <- df_decay$z
t <- df_decay$time

# transform and model
m <- lm(log(z) ~ t)
summary(m)
#>
#> Call:
#> lm(formula = log(z) ~ t)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -0.4479 -0.0993  0.0049  0.0978  0.2802
```

```

#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.6887    0.0306   22.5 <2e-16 ***
#> t            -2.0118    0.0351  -57.3 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.148 on 98 degrees of freedom
#> Multiple R-squared:  0.971, Adjusted R-squared:  0.971
#> F-statistic: 3.28e+03 on 1 and 98 DF, p-value: <2e-16

```

The right panel of [Figure 11-1](#) shows the plot of $\log(z)$ versus time. Superimposed on that plot is their regression line. The fit appears to be much better; this is confirmed by the $R^2 = 0.97$, compared with 0.82 for the linear regression on the original data.

You can embed other functions inside your formula. If you thought the relationship was quadratic, you could use a square-root transformation:

```
lm(sqrt(y) ~ month)
```

You can apply transformations to variables on both sides of the formula, of course. This formula regresses y on the square root of x :

```
lm(y ~ sqrt(x))
```

This regression is for a log-log relationship between x and y :

```
lm(log(y) ~ log(x))
```

See Also

See [Recipe 11.13](#).

11.13 Finding the Best Power Transformation (Box–Cox Procedure)

Problem

You want to improve your linear model by applying a power transformation to the response variable.

Solution

Use the Box–Cox procedure, which is implemented by the `boxcox` function of the MASS package. The procedure will identify a power, λ , such that transforming y into y^λ will improve the fit of your model:

```
library(MASS)
m <- lm(y ~ x)
boxcox(m)
```

Discussion

To illustrate the Box–Cox transformation, let's create some artificial data using the equation $y^{-1.5} = x + \varepsilon$, where ε is an error term:

```
set.seed(9)
x <- 10:100
eps <- rnorm(length(x), sd = 5)
y <- (x + eps) ^ (-1 / 1.5)
```

Then we will (mistakenly) model the data using a simple linear regression and derive an adjusted R^2 of 0.637:

```
m <- lm(y ~ x)
summary(m)
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Residuals:
#>   Min     1Q   Median     3Q    Max
#> -0.04032 -0.01633 -0.00792  0.00996  0.14516
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 0.166885  0.007078   23.6 <2e-16 ***
#> x          -0.001465  0.000116   -12.6 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.0291 on 89 degrees of freedom
#> Multiple R-squared:  0.641, Adjusted R-squared:  0.637
#> F-statistic: 159 on 1 and 89 DF, p-value: <2e-16
```

When plotting the residuals against the fitted values, we get a clue that something is wrong. We can get a `ggplot` residual plot using the `broom` library. The `augment` function from `broom` will put our residuals (and other things) into a data frame for easier plotting. Then we can use `ggplot` to plot:

```
library(broom)
augmented_m <- augment(m)

ggplot(augmented_m, aes(x = .fitted, y = .resid)) +
  geom_point()
```

The result is shown in [Figure 11-2](#).

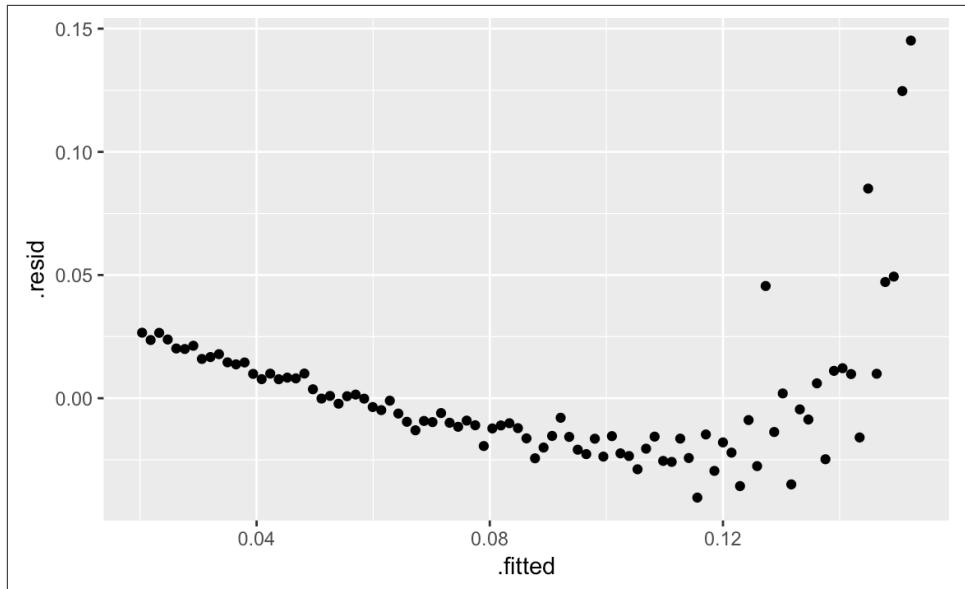


Figure 11-2. Fitted values versus residuals

If you just need a fast peek at the residual plot and don't care if the result is a `ggplot` figure, you can use Base R's `plot` method on the model object, `m`:

```
plot(m, which = 1) # which = 1 plots only the fitted vs. residuals
```

We can see in Figure 11-2 that this plot has a clear parabolic shape. A possible fix is a power transformation on y , so we run the Box–Cox procedure:

```
library(MASS)
#>
## Attaching package: 'MASS'
## The following object is masked from 'package:dplyr':
#>
#>     select
bc <- boxcox(m)
```

The `boxcox` function plots values of λ against the log-likelihood of the resulting model, as shown in Figure 11-3. We want to maximize that log-likelihood, so the function draws a line at the best value and also draws lines at the limits of its confidence interval. In this case, it looks like the best value is around -1.5 , with a confidence interval of about $(-1.75, -1.25)$.

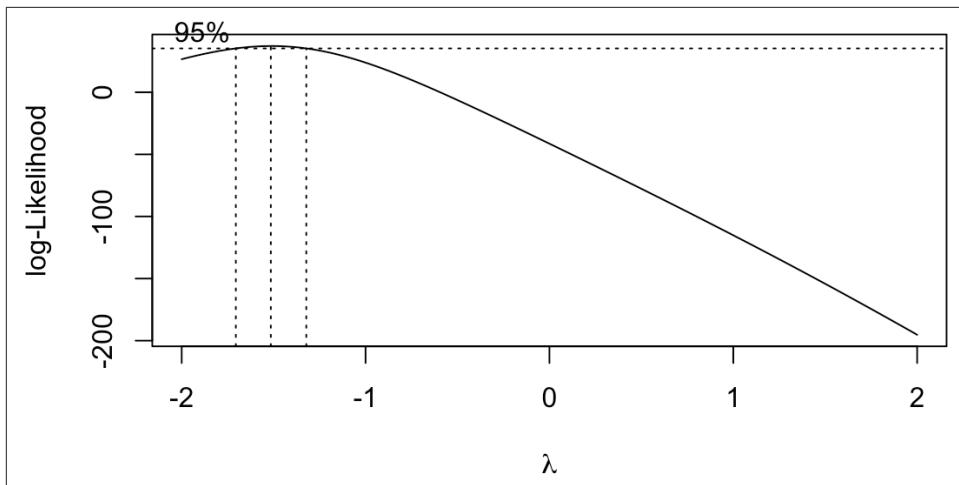


Figure 11-3. Output of `boxcox` on the model (m)

Oddly, the `boxcox` function does not return the best value of λ . Rather, it returns the (x, y) pairs displayed in the plot. It's pretty easy to find the values of λ that yield the largest log-likelihood, y . We use the `which.max` function:

```
which.max(bc$y)
#> [1] 13
```

Then this gives us the position of the corresponding λ :

```
lambda <- bc$x[which.max(bc$y)]
lambda
#> [1] -1.52
```

The function reports that the best λ is -1.52 . In an actual application, we would urge you to interpret this number and choose the power that makes sense to you, rather than blindly accepting this “best” value. Use the graph to assist you in that interpretation. Here, we’ll go with -1.52 .

We can apply the power transform to y and then fit the revised model; this gives a much better R^2 of 0.967:

```
z <- y ^ lambda
m2 <- lm(z ~ x)
summary(m2)
#>
#> Call:
#> lm(formula = z ~ x)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -13.459 -3.711 -0.228  2.206 14.188
#>
```

```

#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.6426    1.2517  -0.51    0.61
#> x            1.0514    0.0205  51.20 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 5.15 on 89 degrees of freedom
#> Multiple R-squared:  0.967, Adjusted R-squared:  0.967
#> F-statistic: 2.62e+03 on 1 and 89 DF, p-value: <2e-16

```

For those who prefer one-liners, the transformation can be embedded right into the revised regression formula:

```
m2 <- lm(I(y ^ lambda) ~ x)
```



By default, `boxcox` searches for values of λ in the range -2 to $+2$. You can change that via the `lambda` argument; see the help page for details.

We suggest viewing the Box–Cox result as a starting point, not as a definitive answer. If the confidence interval for λ includes 1.0, it may be that no power transformation is actually helpful. As always, inspect the residuals before and after the transformation. Did they really improve?

Compare [Figure 11-4](#) (transformed data) with [Figure 11-2](#) (no transformation).

```

augmented_m2 <- augment(m2)

ggplot(augmented_m2, aes(x = .fitted, y = .resid)) +
  geom_point()

```

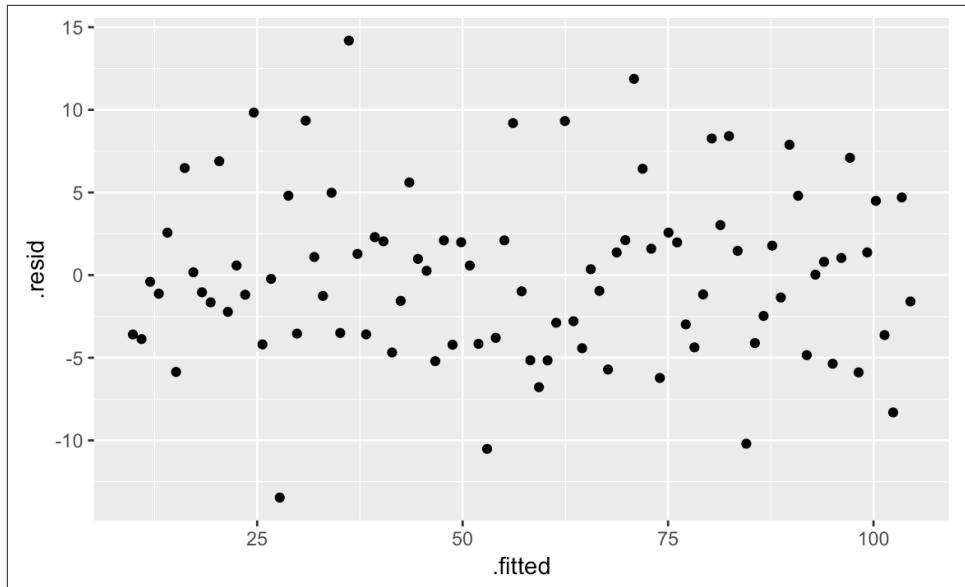


Figure 11-4. Fitted values versus residuals: *m2*

See Also

See [Recipe 11.12](#) and [Recipe 11.16](#).

11.14 Forming Confidence Intervals for Regression Coefficients

Problem

You are performing linear regression and you need the confidence intervals for the regression coefficients.

Solution

Save the regression model in an object; then use the `confint` function to extract confidence intervals:

```
load(file = './data/conf.rdata')
m <- lm(y ~ x1 + x2)
confint(m)
#>              2.5 % 97.5 %
#> (Intercept) -3.90   6.47
#> x1          -2.58   6.24
#> x2          4.67   5.17
```

Discussion

The Solution uses the model $y = \beta_0 + \beta_1(x_1)_i + \beta_2(x_2)_i + \varepsilon_i$. The `confint` function returns the confidence intervals for the intercept (β_0), the coefficient of x_1 (β_1), and the coefficient of x_2 (β_2):

```
confint(m)
#>              2.5 % 97.5 %
#> (Intercept) -3.90   6.47
#> x1          -2.58   6.24
#> x2          4.67   5.17
```

By default, `confint` uses a confidence level of 95%. Use the `level` parameter to select a different level:

```
confint(m, level = 0.99)
#>              0.5 % 99.5 %
#> (Intercept) -5.72   8.28
#> x1          -4.12   7.79
#> x2          4.58   5.26
```

See Also

The `coefplot` function of the `arm` package can plot confidence intervals for regression coefficients.

11.15 Plotting Regression Residuals

Problem

You want a visual display of your regression residuals.

Solution

You can plot the model object by using `broom` to put model results in a data frame, then plot with `ggplot`:

```
m <- lm(y ~ x1 + x2)

library(broom)
augmented_m <- augment(m)

ggplot(augmented_m, aes(x = .fitted, y = .resid)) +
  geom_point()
```

Discussion

Using the linear model `m` from the prior recipe, we can create a simple residual plot:

```
library(broom)
augmented_m <- augment(m)

ggplot(augmented_m, aes(x = .fitted, y = .resid)) +
  geom_point()
```

The output is shown in [Figure 11-5](#).

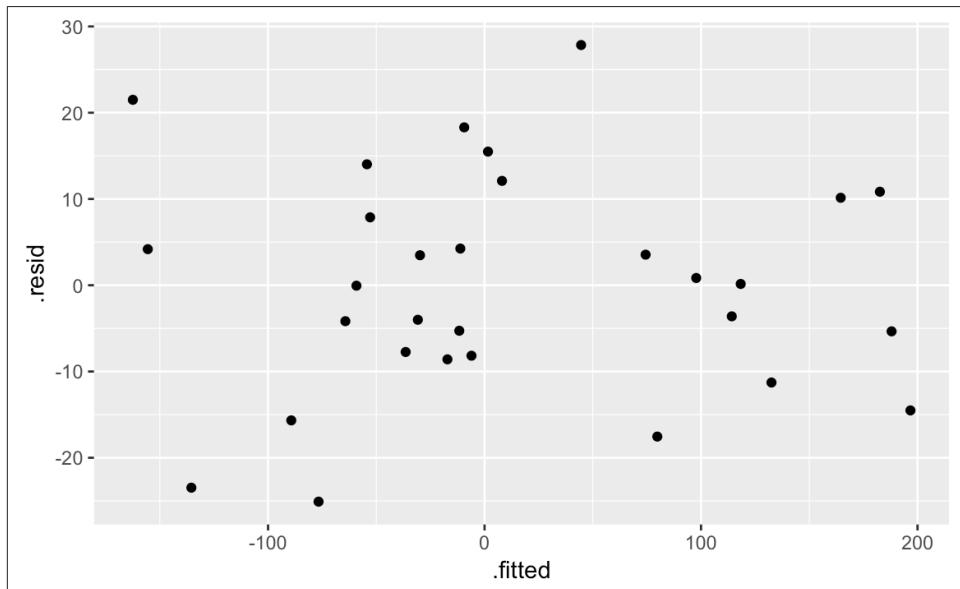


Figure 11-5. Model residual plot

You could also use the Base R `plot` method to get a quick peek, but it will produce Base R graphics output, instead of a `ggplot` graph:

```
plot(m, which = 1)
```

See Also

See [Recipe 11.16](#), which contains examples of residuals plots and other diagnostic plots.

11.16 Diagnosing a Linear Regression

Problem

You have performed a linear regression. Now you want to verify the model's quality by running diagnostic checks.

Solution

Start by plotting the model object, which will produce several diagnostic plots using Base R graphics:

```
m <- lm(y ~ x1 + x2)
plot(m)
```

Next, identify possible outliers either by looking at the diagnostic plot of the residuals or by using the `outlierTest` function of the `car` package:

```
library(car)
outlierTest(m)
```

Finally, identify any overly influential observations. See [Recipe 11.17](#).

Discussion

R fosters the impression that linear regression is easy: just use the `lm` function. Yet fitting the data is only the beginning. It's your job to decide whether the fitted model actually works and works well.

Before anything else, you must have a statistically significant model. Check the F statistic from the model summary ([Recipe 11.4](#)) and be sure that the p -value is small enough for your purposes. Conventionally, it should be less than 0.05 or else your model is likely not very meaningful.

Simply plotting the model object produces several useful diagnostic plots, shown in [Figure 11-6](#):

```
m <- lm(y ~ x1 + x2)
par(mfrow = (c(2, 2))) # this gives us a 2x2 plot
plot(m)
```

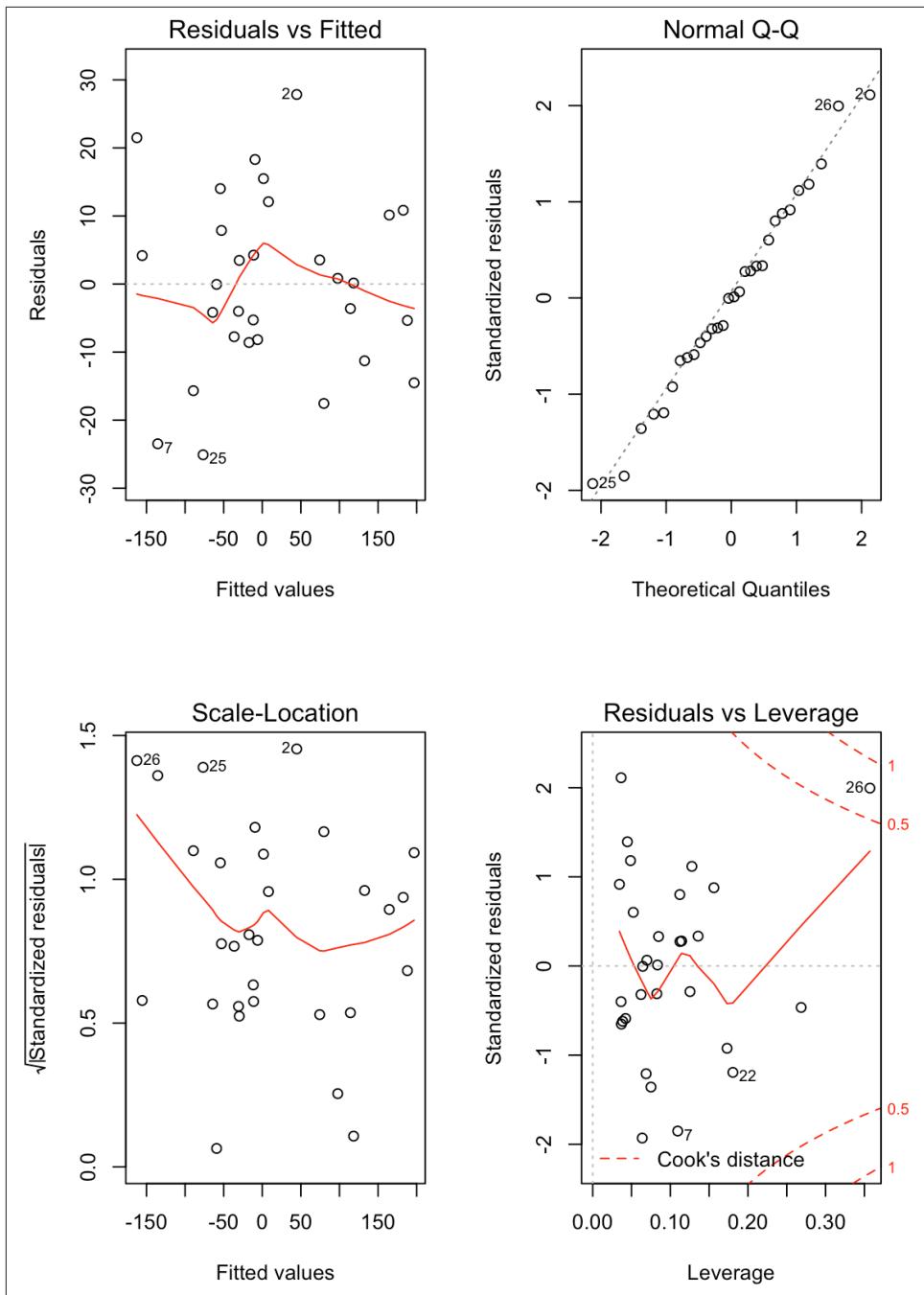


Figure 11-6. Diagnostics of a good fit

Figure 11-6 shows diagnostic plots for a pretty good regression:

- The points in the Residuals vs Fitted plot are randomly scattered with no particular pattern.
- The points in the Normal Q–Q plot are more or less on the line, indicating that the residuals follow a normal distribution.
- In both the Scale–Location plot and the Residuals vs Leverage plot, the points are in a group with none too far from the center.

In contrast, the series of graphs in Figure 11-7 show the diagnostics for a not-so-good regression:

```
load(file = './data/bad.rdata')
m <- lm(y2 ~ x3 + x4)
par(mfrow = c(2, 2))      # this gives us a 2x2 plot
plot(m)
```

Observe that the Residuals vs Fitted plot has a definite parabolic shape. This tells us that the model is incomplete: a quadratic factor is missing that could explain more variation in y . Other patterns in residuals would be suggestive of additional problems: a cone shape, for example, may indicate nonconstant variance in y . Interpreting those patterns is a bit of an art, so we suggest reviewing a good book on linear regression while evaluating the plot of residuals.

There are other problems with these not-so-good diagnostics. The Normal Q–Q plot has more points off the line than it does for the good regression. Both the Scale–Location and Residuals vs Leverage plots show points scattered away from the center, which suggests that some points have excessive leverage.

Another pattern is that point number 28 sticks out in every plot. This warns us that something is odd about that observation. The point could be an outlier, for example. We can check that hunch with the `outlierTest` function of the `car` package:

```
library(car)
outlierTest(m)
#>    rstudent unadjusted p-value Bonferroni p
#> 28     4.46          7.76e-05     0.0031
```

`outlierTest` identifies the model's most outlying observation. In this case, it identified observation number 28 and so confirmed that it could be an outlier.

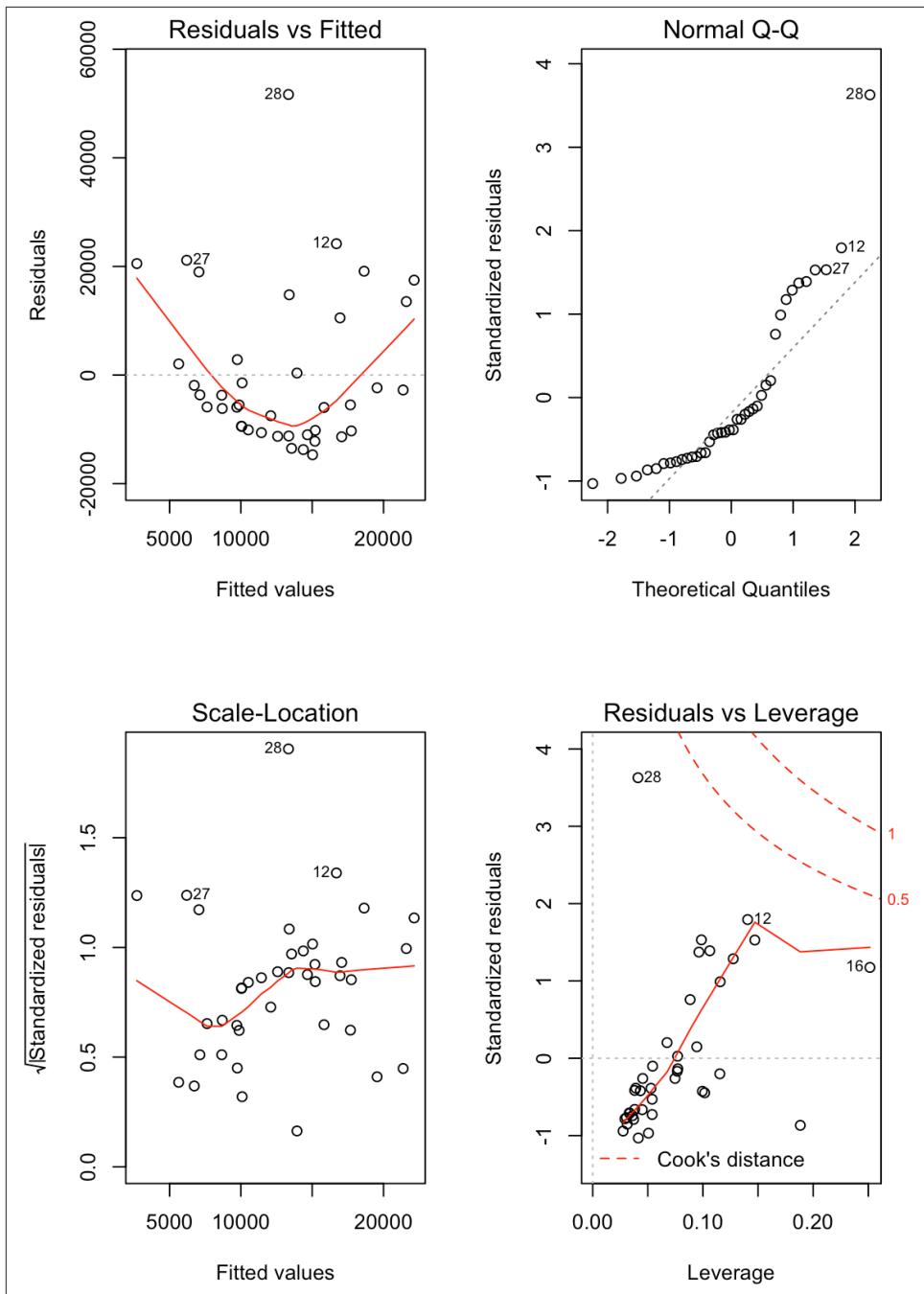


Figure 11-7. Diagnostics of a poor fit

See Also

See [Recipe 11.4](#) and [Recipe 11.17](#). The `car` package is not part of the standard distribution of R; see [Recipe 3.10](#) for how to install it.

11.17 Identifying Influential Observations

Problem

You want to identify the observations that are having the most influence on the regression model. This is useful for diagnosing possible problems with the data.

Solution

The `influence.measures` function reports several useful statistics for identifying influential observations, and it flags the significant ones with an asterisk (*). Its main argument is the model object from your regression:

```
influence.measures(m)
```

Discussion

The title of this recipe could be “Identifying *Overly* Influential Observations,” but that would be redundant. All observations influence the regression model, even if only a little. When a statistician says that an observation is influential, it means that removing the observation would significantly change the fitted regression model. We want to identify those observations because they might be outliers that distort our model; we owe it to ourselves to investigate them.

The `influence.measures` function reports several statistics: DFBETAS, DFFITS, covariance ratio, Cook’s distance, and hat matrix values. If any of these measures indicate that an observation is influential, the function flags that observation with an asterisk (*) along the righthand side:

```
influence.measures(m)
#> Influence measures of
#>   lm(formula = y2 ~ x3 + x4) :
#>
#>     dfb.1_   dfb.x3   dfb.x4   dffit cov.r  cook.d    hat inf
#> 1 -0.18784  0.15174  0.07081 -0.22344 1.059 1.67e-02 0.0506
#> 2  0.27637 -0.04367 -0.39042  0.45416 1.027 6.71e-02 0.0964
#> 3 -0.01775 -0.02786  0.01088 -0.03876 1.175 5.15e-04 0.0772
#> 4  0.15922 -0.14322  0.25615  0.35766 1.133 4.27e-02 0.1156
#> 5 -0.10537  0.00814 -0.06368 -0.13175 1.078 5.87e-03 0.0335
#> 6  0.16942  0.07465  0.42467  0.48572 1.034 7.66e-02 0.1062
#> 7 -0.10128 -0.05936  0.01661 -0.13021 1.078 5.73e-03 0.0333
#> 8 -0.15696  0.04801  0.01441 -0.15827 1.038 8.38e-03 0.0276
#> 9 -0.04582 -0.12089 -0.01032 -0.14010 1.188 6.69e-03 0.0995
```

```

#> 10 -0.01901  0.00624  0.01740 -0.02416 1.147 2.00e-04 0.0544
#> 11 -0.06725 -0.01214  0.04382 -0.08174 1.113 2.28e-03 0.0381
#> 12  0.17580  0.35102  0.62952  0.74889 0.961 1.75e-01 0.1406
#> 13 -0.14288  0.06667  0.06786 -0.15451 1.071 8.04e-03 0.0372
#> 14 -0.02784  0.02366 -0.02727 -0.04790 1.173 7.85e-04 0.0767
#> 15  0.01934  0.03440 -0.01575  0.04729 1.197 7.66e-04 0.0944
#> 16  0.35521 -0.53827 -0.44441  0.68457 1.294 1.55e-01 0.2515 *
#> 17 -0.09184 -0.07199  0.01456 -0.13057 1.089 5.77e-03 0.0381
#> 18 -0.05807 -0.00534 -0.05725 -0.08825 1.119 2.66e-03 0.0433
#> 19  0.00288  0.00438  0.00511  0.00761 1.176 1.99e-05 0.0770
#> 20  0.08795  0.06854  0.19526  0.23490 1.136 1.86e-02 0.0884
#> 21  0.22148  0.42533 -0.33557  0.64699 1.047 1.34e-01 0.1471
#> 22  0.20974 -0.19946  0.36117  0.49631 1.085 8.06e-02 0.1275
#> 23 -0.03333 -0.05436  0.01568 -0.07316 1.167 1.83e-03 0.0747
#> 24 -0.04534 -0.12827 -0.03282 -0.14844 1.189 7.51e-03 0.1016
#> 25 -0.11334  0.00112 -0.05748 -0.13580 1.067 6.22e-03 0.0307
#> 26 -0.23215  0.37364  0.16153 -0.41638 1.258 5.82e-02 0.1883 *
#> 27  0.29815  0.01963 -0.43678  0.51616 0.990 8.55e-02 0.0986
#> 28  0.83069 -0.50577 -0.35404  0.92249 0.303 1.88e-01 0.0411 *
#> 29 -0.09920 -0.07828 -0.02499 -0.14292 1.077 6.89e-03 0.0361
#> # etc.

```

This is the model from [Recipe 11.16](#), where we suspected that observation 28 was an outlier. An asterisk is flagging that observation, confirming that it's overly influential.



This recipe can identify influential observations, but you shouldn't reflexively delete them. Some judgment is required here. Are those observations improving your model or damaging it?

See Also

See [Recipe 11.16](#). Use `help(influence.measures)` to get a list of influence measures and some related functions. See a regression textbook for interpretations of the various influence measures.

11.18 Testing Residuals for Autocorrelation (Durbin–Watson Test)

Problem

You have performed a linear regression and want to check the residuals for autocorrelation.

Solution

The Durbin–Watson test can check the residuals for autocorrelation. The test is implemented by the `dwtest` function of the `lmtest` package:

```
library(lmtest)
m <- lm(y ~ x)           # Create a model object
dwtest(m)                 # Test the model residuals
```

The output includes a p -value. Conventionally, if $p < 0.05$ then the residuals are significantly correlated, whereas $p > 0.05$ provides no evidence of correlation.

You can perform a visual check for autocorrelation by graphing the *autocorrelation function* (ACF) of the residuals:

```
acf(m)                   # Plot the ACF of the model residuals
```

Discussion

The Durbin–Watson test is often used in time series analysis, but it was originally created for diagnosing autocorrelation in regression residuals. Autocorrelation in the residuals is a scourge because it distorts the regression statistics, such as the F statistic and the t statistics for the regression coefficients. The presence of autocorrelation suggests that your model is missing a useful predictor variable or that it should include a time series component, such as a trend or a seasonal indicator.

This first example builds a simple regression model and then tests the residuals for autocorrelation. The test returns a p -value well above zero, which indicates that there is no significant autocorrelation:

```
library(lmtest)
load(file = './data/ac.rdata')
m <- lm(y1 ~ x)
dwtest(m)
#>
#> Durbin-Watson test
#>
#> data: m
#> DW = 2, p-value = 0.4
#> alternative hypothesis: true autocorrelation is greater than 0
```

This second example exhibits autocorrelation in the residuals. The p -value is near zero, so the autocorrelation is likely positive:

```
m <- lm(y2 ~ x)
dwtest(m)
#>
#> Durbin-Watson test
#>
#> data: m
```

```
#> DW = 2, p-value = 0.01  
#> alternative hypothesis: true autocorrelation is greater than 0
```

By default, `dwttest` performs a one-sided test and answers this question: is the autocorrelation of the residuals greater than zero? If your model could exhibit negative autocorrelation (yes, that is possible), then you should use the `alternative` option to perform a two-sided test:

```
dwttest(m, alternative = "two.sided")
```

The Durbin–Watson test is also implemented by the `durbinWatsonTest` function of the `car` package. We suggested the `dwttest` function primarily because we think the output is easier to read.

See Also

Neither the `lmtest` package nor the `car` package is included in the standard distribution of R; see [Recipe 3.8](#) and [Recipe 3.10](#) for accessing their functions and installing them. See [Recipe 14.13](#) and [Recipe 14.16](#) for more regarding tests of autocorrelation.

11.19 Predicting New Values

Problem

You want to predict new values from your regression model.

Solution

Save the predictor data in a data frame. Use the `predict` function, setting the `newdata` parameter to the data frame:

```
load(file = './data/pred2.rdata')  
  
m <- lm(y ~ u + v + w)  
preds <- data.frame(u = 3.1, v = 4.0, w = 5.5)  
predict(m, newdata = preds)  
#> 1  
#> 45
```

Discussion

Once you have a linear model, making predictions is quite easy because the `predict` function does all the heavy lifting. The only annoyance is arranging for a data frame to contain your data.

The `predict` function returns a vector of predicted values with one prediction for every row in the data. The example in the Solution contains one row, so `predict` returned one value.

If your predictor data contains several rows, you get one prediction per row:

```
preds <- data.frame(  
  u = c(3.0, 3.1, 3.2, 3.3),  
  v = c(3.9, 4.0, 4.1, 4.2),  
  w = c(5.3, 5.5, 5.7, 5.9)  
)  
predict(m, newdata = preds)  
#> 1 2 3 4  
#> 43.8 45.0 46.3 47.5
```

In case it's not obvious: the new data needn't contain values for response variables, only predictor variables. After all, you are trying to *calculate* the response, so it would be unreasonable of R to expect you to supply it.

See Also

These are just the point estimates of the predictions. See [Recipe 11.20](#) for the confidence intervals.

11.20 Forming Prediction Intervals

Problem

You are making predictions using a linear regression model. You want to know the prediction intervals: the range of the distribution of the prediction.

Solution

Use the `predict` function and specify `interval = "prediction"`:

```
predict(m, newdata = preds, interval = "prediction")
```

Discussion

This is a continuation of [Recipe 11.19](#), which described packaging your data into a data frame for the `predict` function. We are adding `interval = "prediction"` to obtain prediction intervals.

Here is the example from [Recipe 11.19](#), now with prediction intervals. The new `lwr` and `upr` columns are the lower and upper limits, respectively, for the interval:

```
predict(m, newdata = preds, interval = "prediction")  
#> fit lwr upr  
#> 1 43.8 38.2 49.4  
#> 2 45.0 39.4 50.7  
#> 3 46.3 40.6 51.9  
#> 4 47.5 41.8 53.2
```

By default, `predict` uses a confidence level of 0.95. You can change this via the `level` argument.

A word of caution: these prediction intervals are extremely sensitive to deviations from normality. If you suspect that your response variable is not normally distributed, consider a nonparametric technique, such as the bootstrap (see [Recipe 13.8](#)), for prediction intervals.

11.21 Performing One-Way ANOVA

Problem

Your data is divided into groups, and the groups are normally distributed. You want to know if the groups have significantly different means.

Solution

Use a factor to define the groups. Then apply the `oneway.test` function:

```
oneway.test(x ~ f)
```

Here, `x` is a vector of numeric values and `f` is a factor that identifies the groups. The output includes a *p*-value. Conventionally, a *p*-value of less than 0.05 indicates that two or more groups have significantly different means, whereas a value exceeding 0.05 provides no such evidence.

Discussion

Comparing the means of groups is a common task. One-way ANOVA performs that comparison and computes the probability that they are statistically identical. A small *p*-value indicates that two or more groups likely have different means. (It does *not* indicate that *all* groups have different means.)

The basic ANOVA test assumes that your data has a normal distribution or that, at least, it is pretty close to bell-shaped. If not, use the Kruskal–Wallis test instead (see [Recipe 11.24](#)).

We can illustrate ANOVA with stock market historical data. Is the stock market more profitable in some months than in others? For instance, a common folk myth says that October is a bad month for stock market investors.³ We explored this question by creating a data frame, `GSPC_df`, containing two columns, `r` and `mon`. The factor `r` is

³ In the words of Mark Twain, “October: This is one of the peculiarly dangerous months to speculate in stocks in. The others are July, January, September, April, November, May, March, June, December, August, and February.”

the daily returns in the Standard & Poor's 500 index, a broad measure of stock market performance. The factor `mon` indicates the calendar month in which that change occurred: Jan, Feb, Mar, and so forth. The data covers the period 1950 through 2009.

The one-way ANOVA shows a p -value of 0.03347:

```
load(file = './data/anova.rdata')
oneway.test(r ~ mon, data = GSPC_df)
#>
#> One-way analysis of means (not assuming equal variances)
#>
#> data: r and mon
#> F = 2, num df = 10, denom df = 7000, p-value = 0.03
```

We can conclude that stock market changes varied significantly according to the calendar month.

Before you run to your broker and start flipping your portfolio monthly, however, we should check something: did the pattern change recently? We can limit the analysis to recent data by specifying a `subset` parameter. This works for `oneway.test` just as it does for the `lm` function. The `subset` contains the indexes of observations to be analyzed; all other observations are ignored. Here, we give the indexes of the 2,500 most recent observations, which is about 10 years' worth of data:

```
oneway.test(r ~ mon, data = GSPC_df, subset = tail(seq_along(r), 2500))
#>
#> One-way analysis of means (not assuming equal variances)
#>
#> data: r and mon
#> F = 0.7, num df = 10, denom df = 1000, p-value = 0.8
```

Uh-oh! Those monthly differences evaporated during the past 10 years. The large p -value, 0.8, indicates that changes have not recently varied according to calendar month. Apparently, those differences are a thing of the past.

Notice that the `oneway.test` output says “(not assuming equal variances)”. If you know the groups have equal variances, you'll get a less conservative test by specifying `var.equal = TRUE`:

```
oneway.test(x ~ f, var.equal = TRUE)
```

You can also perform a one-way ANOVA by using the `aov` function like this:

```
m <- aov(x ~ f)
summary(m)
```

However, the `aov` function always assumes equal variances and so is somewhat less flexible than `oneway.test`.

See Also

If the means are significantly different, use [Recipe 11.23](#) to see the actual differences. Use [Recipe 11.24](#) if your data is not normally distributed, as required by ANOVA.

11.22 Creating an Interaction Plot

Problem

You are performing a multiway ANOVA, using two or more categorical variables as predictors. You want a visual check of possible interaction between the predictors.

Solution

Use the `interaction.plot` function:

```
interaction.plot(pred1, pred2, resp)
```

Here, `pred1` and `pred2` are two categorical predictors and `resp` is the response variable.

Discussion

ANOVA is a form of linear regression, so ideally there is a linear relationship between every predictor and the response variable. One source of nonlinearity is an *interaction* between two predictors: as one predictor changes value, the other predictor changes its relationship to the response variable. Checking for interaction between predictors is a basic diagnostic.

The `faraway` package contains a dataset called `rats`. In it, `treat` and `poison` are categorical variables and `time` is the response variable. When plotting `poison` against `time` we are looking for straight, parallel lines, which indicate a linear relationship. However, using the `interaction.plot` function produces [Figure 11-8](#), which reveals that something is not right:

```
library(faraway)
data(rats)
interaction.plot(rats$poison, rats$treat, rats$time)
```

Each line graphs `time` against `poison`. The difference between lines is that each line is for a different value of `treat`. The lines should be parallel, but the top two are not exactly parallel. Evidently, varying the value of `treat` “warped” the lines, introducing a nonlinearity into the relationship between `poison` and `time`.

This signals a possible interaction that we should check. For this data it just so happens that yes, there is an interaction, but no, it is not statistically significant. The

moral is clear: the visual check is useful, but it's not foolproof. Follow up with a statistical check.

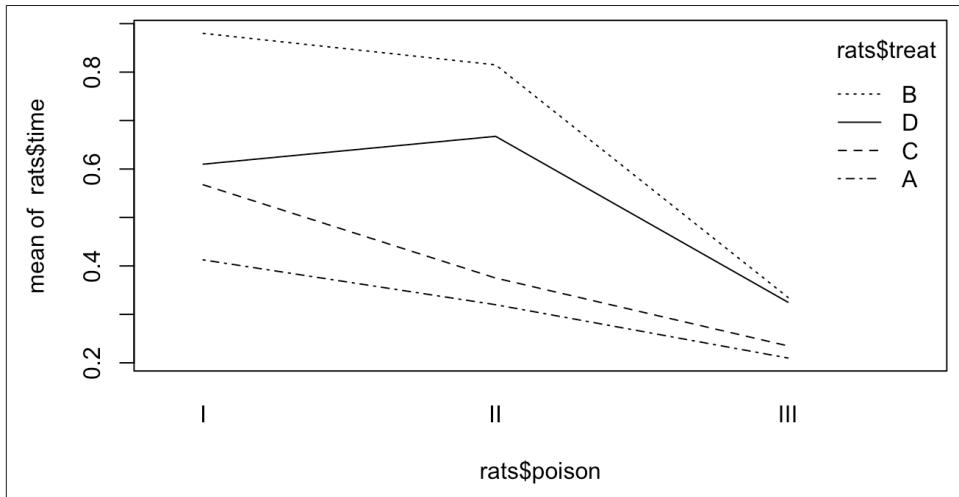


Figure 11-8. Interaction plot

See Also

See [Recipe 11.7](#).

11.23 Finding Differences Between Means of Groups

Problem

Your data is divided into groups, and an ANOVA test indicates that the groups have significantly different means. You want to know the differences between those means for all groups.

Solution

Perform the ANOVA test using the `aov` function, which returns a model object. Then apply the `TukeyHSD` function to the model object:

```
m <- aov(x ~ f)
TukeyHSD(m)
```

Here, `x` is your data and `f` is the grouping factor. You can plot the `TukeyHSD` result to obtain a graphical display of the differences:

```
plot(TukeyHSD(m))
```

Discussion

The ANOVA test is important because it tells you whether or not the groups' means are different. But the test does not identify *which* groups are different, and it does not report their differences.

The TukeyHSD function can calculate those differences and help you identify the largest ones. It uses the “honest significant differences” method invented by John Tukey.

We'll illustrate TukeyHSD by continuing the example from [Recipe 11.21](#), which grouped daily stock market changes by month. Here, we group them by weekday instead, using a factor called `wday` that identifies the day of the week (Mon, ..., Fri) on which the change occurred. We'll use the first 2,500 observations, which roughly cover the period from 1950 to 1960:

```
load(file = './data/anova.rdata')
oneway.test(r ~ wday, subset = 1:2500, data = GSPC_df)
#>
#> One-way analysis of means (not assuming equal variances)
#>
#> data: r and wday
#> F = 10, num df = 4, denom df = 1000, p-value = 5e-10
```

The *p*-value is essentially zero, indicating that average changes varied significantly depending on the weekday. To use the TukeyHSD function, we first perform the ANOVA test using the `aov` function, which returns a model object, and then apply the `TukeyHSD` function to the object:

```
m <- aov(r ~ wday, subset = 1:2500, data = GSPC_df)
TukeyHSD(m)
#> Tukey multiple comparisons of means
#> 95% family-wise confidence level
#>
#> Fit: aov(formula = r ~ wday, data = GSPC_df, subset = 1:2500)
#>
#> $wday
#>          diff      lwr      upr   p adj
#> Mon-Fri -0.003153 -4.40e-03 -0.001911 0.000
#> Thu-Fri -0.000934 -2.17e-03  0.000304 0.238
#> Tue-Fri -0.001855 -3.09e-03 -0.000618 0.000
#> Wed-Fri -0.000783 -2.01e-03  0.000448 0.412
#> Thu-Mon  0.002219  9.79e-04  0.003460 0.000
#> Tue-Mon  0.001299  5.85e-05  0.002538 0.035
#> Wed-Mon  0.002370  1.14e-03  0.003605 0.000
#> Tue-Thu -0.000921 -2.16e-03  0.000314 0.249
#> Wed-Thu  0.000151 -1.08e-03  0.001380 0.997
#> Wed-Tue  0.001072 -1.57e-04  0.002300 0.121
```

Each line in the output table includes the difference between the means of two groups (`diff`) as well as the lower and upper bounds of the confidence interval (`lwr` and `upr`)

for the difference. The first line in the table, for example, compares the Mon group and the Fri group: the difference of their means is 0.003 with a confidence interval of $(-0.0044, -0.0019)$.

Scanning the table, we see that the Wed–Mon comparison had the largest difference, which was 0.00237.

A cool feature of TukeyHSD is that it can display these differences visually, too. Simply plot the function's return value to get output, as shown in [Figure 11-9](#):

```
plot(TukeyHSD(m))
```

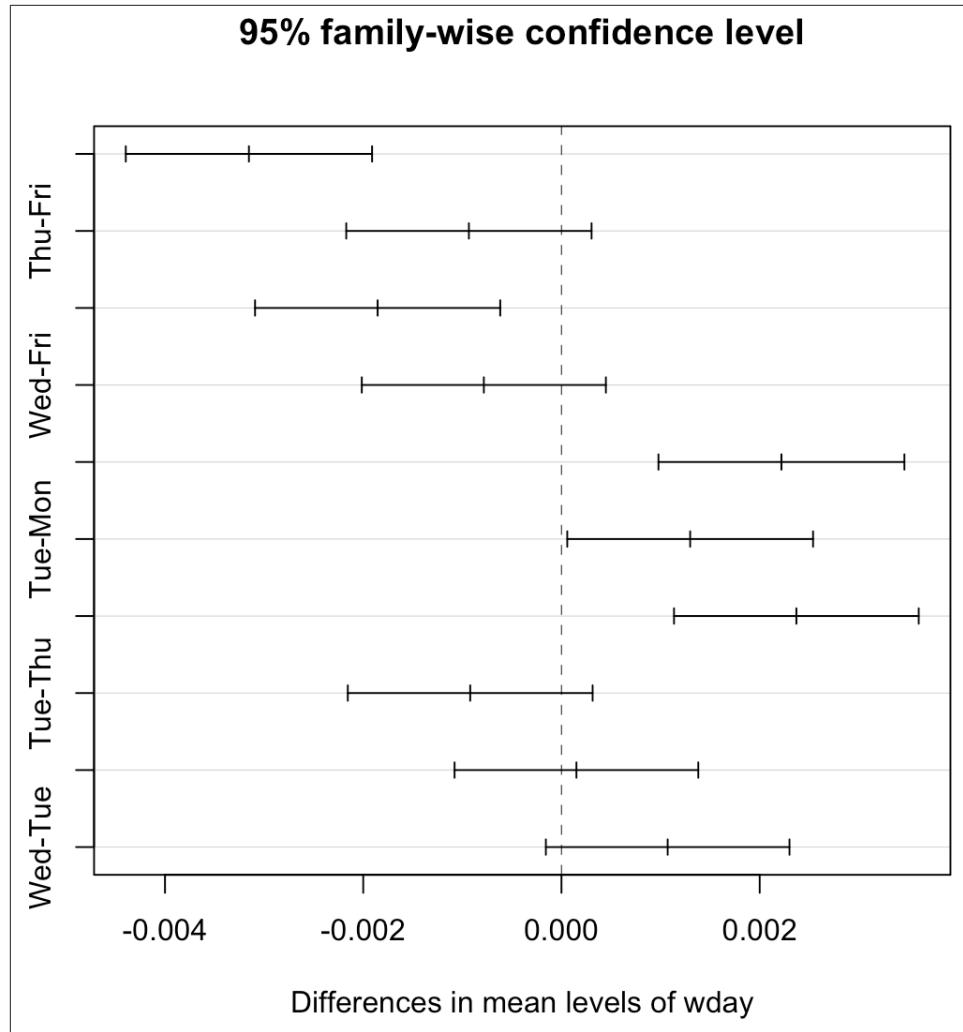


Figure 11-9. TukeyHSD plot

The horizontal lines plot the confidence intervals for each pair. With this visual representation you can quickly see that several confidence intervals cross over zero, indicating that the difference is not necessarily significant. You can also see that the Wed–Mon pair has the largest difference because their confidence interval is farthest to the right.

See Also

See [Recipe 11.21](#).

11.24 Performing Robust ANOVA (Kruskal–Wallis Test)

Problem

Your data is divided into groups. The groups are not normally distributed, but their distributions have similar shapes. You want to perform a test similar to ANOVA—you want to know if the group medians are significantly different.

Solution

Create a factor that defines the groups of your data. Use the `kruskal.test` function, which implements the Kruskal–Wallis test. Unlike the ANOVA test, this test does not depend upon the normality of the data:

```
kruskal.test(x ~ f)
```

Here, `x` is a vector of data and `f` is a grouping factor. The output includes a p -value. Conventionally, $p < 0.05$ indicates that there is a significant difference between the medians of two or more groups, whereas $p > 0.05$ provides no such evidence.

Discussion

Regular ANOVA assumes that your data has a normal distribution. It can tolerate some deviation from normality, but extreme deviations will produce meaningless p -values.

The Kruskal–Wallis test is a nonparametric version of ANOVA, which means that it does not assume normality. However, it does assume same-shaped distributions. You should use the Kruskal–Wallis test whenever your data distribution is nonnormal or simply unknown.

The null hypothesis is that all groups have the same median. Rejecting the null hypothesis (with $p < 0.05$) does not indicate that *all* groups are different, but it does suggest that two or more groups are different.

One year, Paul taught Business Statistics to 94 undergraduate students. The class included a midterm examination, and there were four homework assignments prior to the exam. He wanted to know: what is the relationship between completing the homework and doing well on the exam? If there is no relation, then the homework is irrelevant and needs rethinking.

He created a vector of grades, one per student, and he also created a parallel factor that captured the number of homework assignments completed by that student. The data is in a data frame named `student_data`:

```
load(file = './data/student_data.rdata')
head(student_data)
#> # A tibble: 6 x 4
#>   att.fact hw.mean midterm hw
#>   <fct>    <dbl>    <dbl> <fct>
#> 1 3        0.808    0.818 4
#> 2 3        0.830    0.682 4
#> 3 3        0.444    0.511 2
#> 4 3        0.663    0.670 3
#> 5 2        0.9      0.682 4
#> 6 3        0.948    0.954 4
```

Notice that the `hw` variable—although it appears to be numeric—is actually a factor. It assigns each midterm grade to one of five groups depending upon how many homework assignments the student completed.

The distribution of exam grades is definitely not normal: the students have a wide range of math skills, so there are an unusual number of A and F grades. Hence, regular ANOVA would not be appropriate. Instead we used the Kruskal–Wallis test and obtained a *p*-value of essentially zero (4×10^{-5} , or 0.00004):

```
kruskal.test(midterm ~ hw, data = student_data)
#>
#> Kruskal-Wallis rank sum test
#>
#> data: midterm by hw
#> Kruskal-Wallis chi-squared = 30, df = 4, p-value = 4e-05
```

Obviously, there is a significant performance difference between students who complete their homework and those who do not. But what could Paul actually conclude? At first, he was pleased that the homework appeared so effective. Then it dawned on him that this was a classic error in statistical reasoning: he assumed that correlation implied causality. It does not, of course. Perhaps strongly motivated students do well on both homework and exams, whereas lazy students do not. In that case, the causal factor is degree of motivation, not the brilliance of the homework selection. In the end, he could only conclude something very simple—students who complete the homework will likely do well on the midterm exam—but he still doesn't really know why.

11.25 Comparing Models by Using ANOVA

Problem

You have two models of the same data, and you want to know whether they produce different results.

Solution

The `anova` function can compare two models and report if they are significantly different:

```
anova(m1, m2)
```

Here, `m1` and `m2` are both model objects returned by `lm`. The output from `anova` includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the models are significantly different, whereas a value exceeding 0.05 provides no such evidence.

Discussion

In [Recipe 11.3](#), we used the `anova` function to print the ANOVA table for one regression model. Now we are using the two-argument form to compare two models.

The `anova` function has one strong requirement when comparing two models: one model must be contained within the other. That is, all the terms of the smaller model must appear in the larger model. Otherwise, the comparison is impossible.

The ANOVA analysis performs an F test that is similar to the F test for a linear regression. The difference is that this test is between two models, whereas the regression F test is between using the regression model and using no model.

Suppose we build three models of `y`, adding terms as we go:

```
load(file = './data/anova2.rdata')
m1 <- lm(y ~ u)
m2 <- lm(y ~ u + v)
m3 <- lm(y ~ u + v + w)
```

Is `m2` really different from `m1`? We can use `anova` to compare them, and the result is a p -value of 0.0091:

```
anova(m1, m2)
#> Analysis of Variance Table
#>
#> Model 1: y ~ u
#> Model 2: y ~ u + v
#>   Res.Df RSS Df Sum of Sq    F Pr(>F)
#> 1      18 197
```

```
#> 2      17 130 1      66.4 8.67 0.0091 **
#> ---
#> Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The small p -value indicates that the models are significantly different. Comparing `m2` and `m3`, however, yields a p -value of 0.055:

```
anova(m2, m3)
#> Analysis of Variance Table
#>
#> Model 1: y ~ u + v
#> Model 2: y ~ u + v + w
#>   Res.Df RSS Df Sum of Sq    F Pr(>F)
#> 1      17 130
#> 2      16 103 1      27.5 4.27  0.055 .
#> ---
#> Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This is right on the edge. Strictly speaking, it does not pass our requirement of being smaller than 0.05; however, it's close enough that you might judge the models to be "different enough."

This example is a bit contrived, so it does not show the larger power of `anova`. We use `anova` when, while experimenting with complicated models by adding and deleting multiple terms, we need to know whether or not the new model is really different from the original one. In other words: if we add terms and the new model is essentially unchanged, then the extra terms are not worth the additional complications.

CHAPTER 12

Useful Tricks

The recipes in this chapter are neither obscure numerical calculations nor deep statistical techniques. Yet they are useful functions and idioms that you will likely need at one time or another.

12.1 Peeking at Your Data

Problem

You have a lot of data—too much to display at once. Nonetheless, you want to see some of the data.

Solution

Use `head` to view the first few data values or rows:

```
head(x)
```

Use `tail` to view the last few data values or rows:

```
tail(x)
```

Or you can view the whole thing in an interactive viewer in RStudio:

```
View(x)
```

Discussion

Printing a large dataset is pointless because everything just rolls off your screen. Use `head` to see a little bit of the data (six rows by default):

```
load(file = './data/lab_df.rdata')
head(lab_df)
#>      x lab      y
#> 1 0.0761 NJ 1.621
#> 2 1.4149 KY 10.338
#> 3 2.5176 KY 14.284
#> 4 -0.3043 KY 0.599
#> 5 2.3916 KY 13.091
#> 6 2.0602 NJ 16.321
```

Use `tail` to see the last few rows and the number of rows:

```
tail(lab_df)
#>      x lab      y
#> 195 7.353 KY 38.880
#> 196 -0.742 KY -0.298
#> 197 2.116 NJ 11.629
#> 198 1.606 KY 9.408
#> 199 -0.523 KY -1.089
#> 200 0.675 KY 5.808
```

Both `head` and `tail` allow you to pass a number to the function to set the number of rows returned:

```
tail(lab_df, 2)
#>      x lab      y
#> 199 -0.523 KY -1.09
#> 200 0.675 KY 5.81
```

RStudio comes with an interactive viewer built in. You can call the viewer from the console or a script:

```
View(lab_df)
```

Or you can pipe an object to the viewer:

```
lab_df %>%
  View()
```

When piping to `View` you will notice that the viewer names the View tab simply `.` (just a dot). To get a more informative name, you can put a descriptive name in quotes:

```
lab_df %>%
  View("lab_df test from pipe")
```

The resulting RStudio viewer is shown in [Figure 12-1](#).

The screenshot shows the RStudio viewer window displaying a data frame titled "lab_df test from pipe". The data frame has three columns: "x", "lab", and "y". The "x" column contains numerical values, the "lab" column contains categorical values ("NJ" or "KY"), and the "y" column contains numerical values. The rows are numbered 1 through 13. A summary at the bottom indicates there are 1 to 13 of 200 entries and 3 total columns.

	x	lab	y
1	0.07613317	NJ	1.62128705
2	1.41494855	KY	10.33773006
3	2.51757643	KY	14.28442862
4	-0.30426377	KY	0.59929722
5	2.39156565	KY	13.09133398
6	2.06024789	NJ	16.32053851
7	2.17083546	NJ	12.14102882
8	4.23322043	NJ	23.86045944
9	-0.43771483	NJ	0.40579148
10	4.53473744	KY	24.29519580
11	0.51043681	KY	5.31261990
12	-0.26243714	KY	1.84552135
13	0.56728302	NJ	6.10221163

Showing 1 to 13 of 200 entries, 3 total columns

Figure 12-1. RStudio viewer

See Also

See [Recipe 12.13](#) for seeing the structure of your variable's contents.

12.2 Printing the Result of an Assignment

Problem

You are assigning a value to a variable and you want to see its value.

Solution

Simply put parentheses around the assignment:

```
x <- 1/pi          # Prints nothing  
(x <- 1/pi)       # Prints assigned value  
#> [1] 0.318
```

Discussion

Normally, R inhibits printing when it sees you enter a simple assignment. When you surround the assignment with parentheses, however, it is no longer a simple assignment and so R prints the value. This can be very handy for quick debugging in a script.

See Also

See [Recipe 2.1](#) for more ways to print things.

12.3 Summing Rows and Columns

Problem

You want to sum the rows or columns of a matrix or data frame.

Solution

Use `rowSums` to sum the rows:

```
rowSums(m)
```

Use `colSums` to sum the columns:

```
colSums(m)
```

Discussion

This is a mundane recipe, but it's so common that it deserves mentioning. We use this recipe, for example, when producing reports that include column totals. In this

example, `daily.prod` is a record of this week's factory production and we want totals by product and by day:

```
load(file = './data/daily.prod.rdata')
daily.prod
#> Widgets Gadgets Thingys
#> Mon    179    167    182
#> Tue    153    193    166
#> Wed    183    190    170
#> Thu    153    161    171
#> Fri    154    181    186
colSums(daily.prod)
#> Widgets Gadgets Thingys
#>     822     892     875
rowSums(daily.prod)
#> Mon Tue Wed Thu Fri
#> 528 512 543 485 521
```

These functions return a vector. In the case of column sums, we can append the vector to the matrix and thereby neatly print the data and totals together:

```
rbind(daily.prod, Totals=colSums(daily.prod))
#> Widgets Gadgets Thingys
#> Mon    179    167    182
#> Tue    153    193    166
#> Wed    183    190    170
#> Thu    153    161    171
#> Fri    154    181    186
#> Totals 822    892    875
```

12.4 Printing Data in Columns

Problem

You have several parallel data vectors, and you want to print them in columns.

Solution

Use `cbind` to form the data into columns, then print the result.

Discussion

When you have parallel vectors, it's difficult to see their relationship if you print them separately:

```
load(file = './data/xy.rdata')
print(x)
#> [1] -0.626  0.184 -0.836  1.595  0.330 -0.820  0.487  0.738  0.576 -0.305
print(y)
```

```
#> [1] 1.5118 0.3898 -0.6212 -2.2147 1.1249 -0.0449 -0.0162 0.9438  
#> [9] 0.8212 0.5939
```

Use the `cbind` function to form them into columns that, when printed, show the data's structure:

```
print(cbind(x,y))  
#>           x      y  
#> [1,] -0.626 1.5118  
#> [2,] 0.184 0.3898  
#> [3,] -0.836 -0.6212  
#> [4,] 1.595 -2.2147  
#> [5,] 0.330 1.1249  
#> [6,] -0.820 -0.0449  
#> [7,] 0.487 -0.0162  
#> [8,] 0.738 0.9438  
#> [9,] 0.576 0.8212  
#> [10,] -0.305 0.5939
```

You can include expressions in the output, too. Use a tag to give them a column heading:

```
print(cbind(x, y, Total = x + y))  
#>           x      y   Total  
#> [1,] -0.626 1.5118 0.885  
#> [2,] 0.184 0.3898 0.573  
#> [3,] -0.836 -0.6212 -1.457  
#> [4,] 1.595 -2.2147 -0.619  
#> [5,] 0.330 1.1249 1.454  
#> [6,] -0.820 -0.0449 -0.865  
#> [7,] 0.487 -0.0162 0.471  
#> [8,] 0.738 0.9438 1.682  
#> [9,] 0.576 0.8212 1.397  
#> [10,] -0.305 0.5939 0.289
```

12.5 Binning Your Data

Problem

You have a vector, and you want to split the data into groups according to intervals. Statisticians call this *binning* your data.

Solution

Use the `cut` function. You must define a vector, say `breaks`, that gives the ranges of the intervals. The `cut` function will group your data according to those intervals. It returns a factor whose levels (elements) identify each datum's group:

```
f <- cut(x, breaks)
```

Discussion

This example generates 1,000 random numbers that have a standard normal distribution. It breaks them into six groups by defining intervals at ± 1 , ± 2 , and ± 3 standard deviations:

```
x <- rnorm(1000)
breaks <- c(-3, -2, -1, 0, 1, 2, 3)
f <- cut(x, breaks)
```

The result is a factor, f, that identifies the groups. The `summary` function shows the number of elements by level. R creates names for each level, using the mathematical notation for an interval:

```
summary(f)
#> (-3, -2] (-2, -1] (-1, 0] (0, 1] (1, 2] (2, 3] NA's
#>     25      147      341      332      132       18      5
```

The results are bell-shaped, which is what we expect from the `rnorm` function. There are five NA values, indicating that two values in x fell outside the defined intervals.

We can use the `labels` parameter to give nice, predefined names to the six groups instead of the funky synthesized names:

```
f <- cut(x, breaks, labels = c("Bottom", "Low", "Neg", "Pos", "High", "Top"))
```

Now the `summary` function uses our names:

```
summary(f)
#> Bottom   Low    Neg    Pos    High   Top    NA's
#>     25     147    341    332    132     18     5
```

Binning is useful for summaries such as histograms. But it results in information loss, which can be harmful in modeling. Consider the extreme case of binning a continuous variable into two values, high and low. The binned data has only two possible values, so you have replaced a rich source of information with *one bit* of information. Where the continuous variable might be a powerful predictor, the binned variable can distinguish at most two states and so will likely have only a fraction of the original power. Before you bin, we suggest exploring other transformations that are less lossy.

12.6 Finding the Position of a Particular Value

Problem

You have a vector. You know a particular value occurs in the contents, and you want to know its position.

Solution

The `match` function will search a vector for a particular value and return the position:

```
vec <- c(100, 90, 80, 70, 60, 50, 40, 30, 20, 10)
match(80, vec)
#> [1] 3
```

Here `match` returns 3, which is the position of 80 within `vec`.

Discussion

There are special functions for finding the location of the minimum and maximum values—which `.min` and `.which.max`, respectively:

```
vec <- c(100,90,80,70,60,50,40,30,20,10)
which.min(vec)           # Position of smallest element
#> [1] 10
which.max(vec)           # Position of largest element
#> [1] 1
```

See Also

This technique is used in [Recipe 11.13](#).

12.7 Selecting Every nth Element of a Vector

Problem

You want to select every n th element of a vector.

Solution

Create a logical indexing vector that is TRUE for every n th element. One approach is to find all subscripts that equal zero when taken modulo n :

```
v[seq_along(v) %% n == 0]
```

Discussion

This problem arises in systematic sampling: we want to sample a dataset by selecting every n th element. The `seq_along(v)` function generates the sequence of integers that can index `v`; it is equivalent to `1:length(v)`. We compute each index value modulo n by the expression:

```
v <- rnorm(10)
n <- 2
seq_along(v) %% n
#> [1] 1 0 1 0 1 0 1 0 1 0
```

Then we find those values that equal zero:

```
seq_along(v) %% n == 0
#> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
```

The result is a logical vector, the same length as v and with TRUE at every n th element, that can index v to select the desired elements:

```
v
#> [1] 2.325 0.524 0.971 0.377 -0.996 -0.597 0.165 -2.928 -0.848 0.799
v[ seq_along(v) %% n == 0 ]
#> [1] 0.524 0.377 -0.597 -2.928 0.799
```

If you just want something simple like every second element, you can use the Recycling Rule in a clever way. Index v with a two-element logical vector, like this:

```
v[c(FALSE, TRUE)]
#> [1] 0.524 0.377 -0.597 -2.928 0.799
```

If v has more than two elements, then the indexing vector is too short. Hence, R will invoke the Recycling Rule and expand the index vector to the length of v , recycling its contents. That gives an index vector that is FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, and so forth. Voilà! The final result is every second element of v .

See Also

See [Recipe 5.3](#) for more about the Recycling Rule.

12.8 Finding Minimums or Maximums

Problem

You have two vectors, v and w , and you want to find the minimums or the maximums of pairwise elements. That is, you want to calculate:

$$\min(v_1, w_1), \min(v_2, w_2), \min(v_3, w_3), \dots$$

or:

$$\max(v_1, w_1), \max(v_2, w_2), \max(v_3, w_3), \dots$$

Solution

R calls these the *parallel minimum* and the *parallel maximum*. The calculation is performed by `pmin(v,w)` and `pmax(v,w)`, respectively:

```
pmin(1:5, 5:1)    # Find the element-by-element minimum
#> [1] 1 2 3 2 1
pmax(1:5, 5:1)    # Find the element-by-element maximum
#> [1] 5 4 3 4 5
```

Discussion

When an R beginner wants pairwise minimums or maximums, a common mistake is to write `min(v,w)` or `max(v,w)`. Those are not pairwise operations: `min(v,w)` returns a single value, the minimum over all `v` and `w`. Likewise, `max(v,w)` returns a single value from all of `v` and `w`.

The `pmin` and `pmax` values compare their arguments in parallel, picking the minimum or maximum for each subscript. They return a vector that matches the length of the inputs.

You can combine `pmin` and `pmax` with the Recycling Rule to perform useful hacks. Suppose the vector `v` contains both positive and negative values, and you want to reset the negative values to zero. This does the trick:

```
v <- c(-3:3)
v
#> [1] -3 -2 -1  0  1  2  3
v <- pmax(v, 0)
v
#> [1] 0 0 0 0 1 2 3
```

By the Recycling Rule, R expands the zero-valued scalar into a vector of zeros that is the same length as `v`. Then `pmax` does an element-by-element comparison, taking the larger of zero and each element of `v`.

Actually, `pmin` and `pmax` are more powerful than the Solution indicates. They can take more than two vectors, comparing all vectors in parallel.

It is not uncommon to use `pmin` or `pmax` to calculate a new variable in a data frame based on multiple fields. Let's look at a quick example:

```
df <- data.frame(a = c(1,5,8),
                  b = c(2,3,7),
                  c = c(0,4,9))
df %>%
  mutate(max_val = pmax(a,b,c))
#>   a b c max_val
#> 1 1 2 0      2
#> 2 5 3 4      5
#> 3 8 7 9      9
```

We can see the new column, `max_val`, now contains the row-by-row max value from the three input columns.

See Also

See [Recipe 5.3](#) for more about the Recycling Rule.

12.9 Generating All Combinations of Several Variables

Problem

You have two or more variables. You want to generate all combinations of their levels, also known as their *Cartesian product*.

Solution

Use the `expand.grid` function. Here, `f` and `g` are vectors:

```
expand.grid(f, g)
```

Discussion

This code snippet creates two vectors—`sides` represents the two sides of a coin, and `faces` represents the six faces of a die (those little spots on a die are called *pips*):

```
sides <- c("Heads", "Tails")
faces <- c("1 pip", paste(2:6, "pips"))
```

We can use `expand.grid` to find all combinations of one roll of the die and one coin toss:

```
expand.grid(faces, sides)
#>      Var1  Var2
#> 1   1 pip Heads
#> 2   2 pips Heads
#> 3   3 pips Heads
#> 4   4 pips Heads
#> 5   5 pips Heads
#> 6   6 pips Heads
#> 7   1 pip Tails
#> 8   2 pips Tails
#> 9   3 pips Tails
#> 10  4 pips Tails
#> 11  5 pips Tails
#> 12  6 pips Tails
```

Similarly, we could find all combinations of two dice, but we won't print the output here because it's 36 lines long:

```
expand.grid(faces, faces)
```

The result of `expand.grid` is a data frame. R automatically provides the row names and column names.

The Solution and the example show the Cartesian product of two vectors, but `expand.grid` can handle three or more factors, too.

See Also

If you’re working with strings and want a bit more control over how you bring the combinations together, then you can also use [Recipe 7.6](#) to generate combinations.

12.10 Flattening a Data Frame

Problem

You have a data frame of numeric values. You want to process all its elements together, not as separate columns—for example, to find the mean across all values.

Solution

Convert the data frame to a matrix and then process the matrix. This example finds the mean of all elements in the data frame `dfrm`:

```
mean(as.matrix(dfrm))
```

It is sometimes necessary then to convert the matrix to a vector. In that case, use `as.vector(as.matrix(dfrm))`.

Discussion

Suppose we have a data frame, such as the factory production data from [Recipe 12.3](#):

```
load(file = './data/daily.prod.rdata')
daily.prod
#>     Widgets Gadgets Thingys
#> Mon     179     167     182
#> Tue     153     193     166
#> Wed     183     190     170
#> Thu     153     161     171
#> Fri     154     181     186
```

Suppose also that we want the average daily production across all days and products. This won’t work:

```
mean(daily.prod)
#> Warning in mean.default(daily.prod): argument is not numeric or logical:
#> returning NA
#> [1] NA
```

The `mean` function doesn’t really know what to do with a data frame, so it just throws an error. When you want the average across all values, first collapse the data frame down to a matrix:

```
mean(as.matrix(daily.prod))
#> [1] 173
```

This recipe works only on data frames with all-numeric data. Recall that converting a data frame with mixed data (numeric columns mixed with character columns or factors) into a matrix forces all columns to be converted to characters.

See Also

See [Recipe 5.29](#) for more about converting between data types.

12.11 Sorting a Data Frame

Problem

You have a data frame. You want to sort the contents, using one column as the sort key.

Solution

Use the `arrange` function from the `dplyr` package:

```
df <- arrange(df, key)
```

Here `df` is a data frame and `key` is the sort-key column.

Discussion

The `sort` function is great for vectors but is ineffective for data frames. Suppose we have the following data frame and we want to sort by month:

```
load(file = './data/outcome.rdata')
print(df)
#>   month day outcome
#> 1     7  11     Win
#> 2     8  10    Lose
#> 3     8  25     Tie
#> 4     6  27     Tie
#> 5     7  22     Win
```

The `arrange` function rearranges the months into ascending order and returns the entire data frame:

```
library(dplyr)
arrange(df, month)
#>   month day outcome
#> 1     6  27     Tie
#> 2     7  11     Win
#> 3     7  22     Win
#> 4     8  10    Lose
#> 5     8  25     Tie
```

After rearranging the data frame, the month column is in ascending order—just as we wanted. If you want to sort the data in descending order, put a - in front of the column you want to sort by:

```
arrange(df, -month)
#>   month day outcome
#> 1     8   10    Lose
#> 2     8   25     Tie
#> 3     7   11     Win
#> 4     7   22     Win
#> 5     6   27     Tie
```

If you want to sort by multiple columns, you can add them to the `arrange` function. The following example sorts by month first, then by day:

```
arrange(df, month, day)
#>   month day outcome
#> 1     6   27     Tie
#> 2     7   11     Win
#> 3     7   22     Win
#> 4     8   10    Lose
#> 5     8   25     Tie
```

Within months 7 and 8, the days are now sorted into ascending order.

12.12 Stripping Attributes from a Variable

Problem

A variable is carrying around old attributes. You want to remove some or all of them.

Solution

To remove all attributes, assign `NULL` to the variable's `attributes` property:

```
attributes(x) <- NULL
```

To remove a single attribute, select the attribute using the `attr` function, and set it to `NULL`:

```
attr(x, "attributeName") <- NULL
```

Discussion

Any variable in R can have attributes. An attribute is simply a name/value pair, and the variable can have many of them. A common example is the dimensions of a matrix variable, which are stored in an attribute. The attribute name is `dim` and the attribute value is a two-element vector giving the number of rows and columns.

You can view the attributes of `x` by printing `attributes(x)` or `str(x)`.

Sometimes you want just a number and R insists on giving it attributes. This can happen when you fit a simple linear model and extract the slope, which is the second regression coefficient:

```
load(file = './data/conf.rdata')
m <- lm(y ~ x1)
slope <- coef(m)[2]
slope
#> x1
#> -11
```

When we print `slope`, R also prints "x1". That is a name attribute given by `lm` to the coefficient (because it's the coefficient for the `x1` variable). We can see that more clearly by printing the internals of `slope`, which reveals a "names" attribute:

```
str(slope)
#> Named num -11
#> - attr(*, "names")= chr "x1"
```

It's easy to strip out all the attributes, after which the slope value becomes simply a number:

```
attributes(slope) <- NULL      # Strip off all attributes
str(slope)                  # Now the "names" attribute is gone
#> num -11

slope                      # And the number prints cleanly without a label
#> [1] -11
```

Alternatively, we could have stripped out the single offending attribute this way:

```
attr(slope, "names") <- NULL
```



Remember that a matrix is a vector (or list) with a `dim` attribute. If you strip out all the attributes from a matrix, that will strip away the dimensions and thereby turn it into a mere vector (or list). Furthermore, stripping the attributes from an object (specifically, an S3 object) can render it useless. So, remove attributes with care.

See Also

See [Recipe 12.13](#) for more about seeing attributes.

12.13 Revealing the Structure of an Object

Problem

You called a function that returned something. Now you want to look inside that something and learn more about it.

Solution

Use `class` to determine the thing's object class:

```
class(x)
```

Use `mode` to strip away the object-oriented features and reveal the underlying structure:

```
mode(x)
```

Use `str` to show the internal structure and contents:

```
str(x)
```

Discussion

We are regularly amazed by how often we call a function, get something back, and wonder: “What the heck is this thing?” Theoretically, the function documentation should explain the returned value, but somehow we feel better when we can see its structure and contents ourselves. This is especially true for objects with a nested structure: objects within objects.

Let’s dissect the value returned by `lm` (the linear modeling function) in the simplest linear regression recipe, [Recipe 11.1](#):

```
load(file = './data/conf.rdata')
m <- lm(y ~ x1)
print(m)
#>
#> Call:
#> lm(formula = y ~ x1)
#>
#> Coefficients:
#> (Intercept)           x1
#>       15.9             -11.0
```

Always start by checking the thing’s class. The class indicates if it’s a vector, matrix, list, data frame, or object:

```
class(m)
#> [1] "lm"
```

Hmmm. It seems that `m` is an object of class `lm`. That may not mean anything to you but we know that all object classes are built upon the native data structures (vector, matrix, list, or data frame). We can use `mode` to strip away the object facade and reveal the underlying structure:

```
mode(m)
#> [1] "list"
```

Ah-ha! It seems that `m` is built on a list structure. Now we can use list functions and operators to dig into its contents. First, we want to know the names of its list elements:

```
names(m)
#> [1] "coefficients"   "residuals"      "effects"       "rank"
#> [5] "fitted.values"  "assign"        "qr"           "df.residual"
#> [9] "xlevels"         "call"          "terms"        "model"
```

The first list element is called "`coefficients`". We could guess those are the regression coefficients. Let's have a look:

```
m$coefficients
#> (Intercept)      x1
#>      15.9      -11.0
```

Yes, that's what they are. We recognize those values.

We could continue digging into the list structure of `m`, but that would get tedious. The `str` function does a good job of revealing the internal structure of any variable:

```
str(m)
#> List of 12
#> $ coefficients : Named num [1:2] 15.9 -11
#> ... attr(*, "names")= chr [1:2] "(Intercept)" "x1"
#> $ residuals    : Named num [1:30] 36.6 58.6 112.1 -35.2 -61.7 ...
#> ... attr(*, "names")= chr [1:30] "1" "2" "3" "4" ...
#> $ effects      : Named num [1:30] -73.1 69.3 93.9 -31.1 -66.3 ...
#> ... attr(*, "names")= chr [1:30] "(Intercept)" "x1" "" ""
#> $ rank         : int 2
#> $ fitted.values: Named num [1:30] 25.69 13.83 -1.55 28.25 16.74 ...
#> ... attr(*, "names")= chr [1:30] "1" "2" "3" "4" ...
#> $ assign        : int [1:2] 0 1
#> $ qr            :List of 5
#> ... $ qr      : num [1:30, 1:2] -5.477 0.183 0.183 0.183 0.183 ...
#> ... .attr(*, "dimnames")=List of 2
#> ... ... $ : chr [1:30] "1" "2" "3" "4" ...
#> ... ... $ : chr [1:2] "(Intercept)" "x1"
#> ... .attr(*, "assign")= int [1:2] 0 1
#> ... $ qraux: num [1:2] 1.18 1.02
#> ... $ pivot: int [1:2] 1 2
#> ... $ tol  : num 1e-07
#> ... $ rank : int 2
#> ... .attr(*, "class")= chr "qr"
#> $ df.residual : int 28
#> $ xlevels     : Named list()
#> $ call         : language lm(formula = y ~ x1)
#> $ terms        :Classes 'terms', 'formula' language y ~ x1
#> ... .attr(*, "variables")= language list(y, x1)
#> ... .attr(*, "factors")= int [1:2, 1] 0 1
#> ... ... .attr(*, "dimnames")=List of 2
#> ... ... ... $ : chr [1:2] "y" "x1"
#> ... ... ... $ : chr "x1"
```

```

#> ... .-. attr(*, "term.labels")= chr "x1"
#> ... .-. attr(*, "order")= int 1
#> ... .-. attr(*, "intercept")= int 1
#> ... .-. attr(*, "response")= int 1
#> ... .-. attr(*, ".Environment")=<environment: R_GlobalEnv>
#> ... .-. attr(*, "predvars")= language list(y, x1)
#> ... .-. attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
#> ... .-. attr(*, "names")= chr [1:2] "y" "x1"
#> $ model      : 'data.frame': 30 obs. of 2 variables:
#> ..$ y : num [1:30] 62.25 72.45 110.59 -6.94 -44.99 ...
#> ..$ x1: num [1:30] -0.8969 0.1848 1.5878 -1.1304 -0.0803 ...
#> ... .attr(*, "terms")=Classes 'terms', 'formula' language y ~ x1
#> ... .-. attr(*, "variables")= language list(y, x1)
#> ... .-. attr(*, "factors")= int [1:2, 1] 0 1
#> ... .-. attr(*, "dimnames")=List of 2
#> ... .-. .$. : chr [1:2] "y" "x1"
#> ... .-. .$. : chr "x1"
#> ... .-. attr(*, "term.labels")= chr "x1"
#> ... .-. attr(*, "order")= int 1
#> ... .-. attr(*, "intercept")= int 1
#> ... .-. attr(*, "response")= int 1
#> ... .-. attr(*, ".Environment")=<environment: R_GlobalEnv>
#> ... .-. attr(*, "predvars")= language list(y, x1)
#> ... .-. attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
#> ... .-. attr(*, "names")= chr [1:2] "y" "x1"
#> - attr(*, "class")= chr "lm"

```

Notice that `str` shows all the elements of `m` and then recursively dumps each element's contents and attributes. Long vectors and lists are truncated to keep the output manageable.

There is an art to exploring an R object. Use `class`, `mode`, and `str` to dig through the layers. We have found that often `str` tells you everything you want to know...and sometimes a lot more!

12.14 Timing Your Code

Problem

You want to know how much time is required to run your code. This is useful, for example, when you are optimizing your code and need “before” and “after” numbers to measure the improvement.

Solution

The `tictoc` package contains a very easy way to time and label chunks of code. The `tic` function starts a timer and the `toc` function stops the timer and reports the execution time:

```
library(tictoc)
tic('Optional helpful name here')
aLongRunningExpression()
toc()
```

The output is the execution time in seconds.

Discussion

Suppose we want to know the time required to generate 10,000,000 random normal numbers and sum them together:

```
library(tictoc)
tic('making big numbers')
total_val <- sum(rnorm(1e7))
toc()
#> making big numbers: 0.794 sec elapsed
```

The `toc` function returns the message set in `tic` along with the runtime in seconds.

If you assign the result of `toc` to an object, you can have access to the underlying start time, finish time, and message:

```
tic('two sums')
sum(rnorm(10000000))
#> [1] -84.1
sum(rnorm(10000000))
#> [1] -3899
toc_result <- toc()
#> two sums: 1.373 sec elapsed

print(toc_result)
#> $tic
#> elapsed
#>   2.64
#>
#> $toc
#> elapsed
#>   4.01
#>
#> $msg
#> [1] "two sums"
```

If you want to report the results in minutes (or hours!), you can use the elements of the output to get at the underlying start and finish times:

```
print(paste('the code ran in',
            round((toc_result$toc - toc_result$tic) / 60, 4),
            'minutes'))
#> [1] "the code ran in 0.0229 minutes"
```

You can accomplish the same thing using just `Sys.time` calls, but without the convenience of labeling and clarity of syntax provided by `toctoc`:

```
start <- Sys.time()
sum(rnorm(10000000))
#> [1] 3607
sum(rnorm(10000000))
#> [1] 1893
Sys.time() - start
#> Time difference of 1.37 secs
```

12.15 Suppressing Warnings and Error Messages

Problem

A function is producing annoying error messages or warning messages. You don't want to see them.

Solution

Surround the function call with `suppressMessage(...)` or `suppressWarnings(...)`:

```
suppressMessage(annoyingFunction())
suppressWarnings(annoyingFunction())
```

Discussion

The Augmented Dickey–Fuller Test, `adf.test`, is a popular time series function. However, it produces an annoying warning message, shown here at the bottom of the output, when the *p*-value is below 0.01:

```
library(tseries)
load(file = './data/adf.rdata')
results <- adf.test(x)
#> Warning in adf.test(x): p-value smaller than printed p-value
```

Fortunately, we can muzzle the function by calling it inside `suppressWarnings(...)`:

```
results <- suppressWarnings(adf.test(x))
```

Notice that the warning message disappeared. The message is not entirely lost because R retains it internally. We can retrieve the message at our leisure by using the `warnings` function:

```
warnings()
```

Some functions also produce “messages” (in R terminology), which are even more benign than warnings. Typically, they are merely informative and not signals of problems. If such a message is annoying you, you can make it disappear by calling the function inside `suppressMessages(...)`.

See Also

See the `options` function for other ways to control the reporting of errors and warnings.

12.16 Taking Function Arguments from a List

Problem

Your data is captured in a list structure. You want to pass the data to a function, but the function does not accept a list.

Solution

In simple cases, convert the list to a vector. For more complex cases, the `do.call` function can break the list into individual arguments and call your function:

```
do.call(function, list)
```

Discussion

If your data is in a vector, life is simple and most R functions work as expected:

```
vec <- c(1, 3, 5, 7, 9)
mean(vec)
#> [1] 5
```

If your data is captured in a list, some functions complain and return a useless result, like this:

```
numbers <- list(1, 3, 5, 7, 9)
mean(numbers)
#> Warning in mean.default(numbers): argument is not numeric or logical:
#> returning NA
#> [1] NA
```

The `numbers` list is a simple, one-level list, so we can just convert it to a vector and call the function:

```
mean(unlist(numbers))
#> [1] 5
```

The big headaches come when you have multilevel list structures: lists within lists. These can occur within complex data structures. Here is a list of lists in which each sublist is a column of data:

```
my_lists <-
  list(col1 = list(7, 8),
       col2 = list(70, 80),
       col3 = list(700, 800))
```

```

my_lists
#> $col1
#> $col1[[1]]
#> [1] 7
#>
#> $col1[[2]]
#> [1] 8
#>
#>
#> $col2
#> $col2[[1]]
#> [1] 70
#>
#> $col2[[2]]
#> [1] 80
#>
#>
#> $col3
#> $col3[[1]]
#> [1] 700
#>
#> $col3[[2]]
#> [1] 800

```

Suppose we want to form this data into a matrix. The `cbind` function is supposed to create data columns, but it gets confused by the list structure and returns something useless:

```

cbind(my_lists)
#>      my_lists
#> col1 List,2
#> col2 List,2
#> col3 List,2

```

If we `unlist` the data then we just get one big, long column, which is not what we are after either:

```

cbind(unlist(my_lists))
#>      [,1]
#> col11    7
#> col12    8
#> col21   70
#> col22   80
#> col31  700
#> col32  800

```

The solution is to use `do.call`, which splits the list into individual items and then calls `cbind` on those items:

```

do.call(cbind, my_lists)
#>      col1 col2 col3
#> [1,] 7    70   700
#> [2,] 8    80   800

```

Using `do.call` in that way is functionally identical to calling `cbind` like this:

```
cbind(my_lists[[1]], my_lists[[2]], my_lists[[3]])
#>      [,1] [,2] [,3]
#> [1,]    7    70   700
#> [2,]    8    80   800
```



Be careful if the list elements have names. In that case, `do.call` interprets the element names as names of parameters to the function, which might cause trouble.

This recipe presents the most basic use of `do.call`. The function is quite powerful and has many other uses. See the help page for more details.

See Also

See [Recipe 5.29](#) for converting between data types.

12.17 Defining Your Own Binary Operators

Problem

You want to define your own binary operators, making your R code more streamlined and readable.

Solution

R recognizes any text between percent signs (%...%) as a binary operator. Create and define a new binary operator by assigning a two-argument function to it.

Discussion

R contains an interesting feature that lets you define your own binary operators. Any text between two percent signs (%...%) is automatically interpreted by R as a binary operator. R predefines several such operators, such as %/% for integer division, %*% for matrix multiplication, and the pipe %>% in the `magrittr` package.

You can create a new binary operator by assigning a function to it. This example creates an operator, %+-%:

```
'%+-%' <- function(x, margin)
  x + c(-1, +1) * margin
```

The expression `x %+-% m` calculates $x \pm m$. Here it calculates $100 \pm (1.96 \times 15)$, the two-standard-deviation range of a standard IQ test:

```
100 %+-% (1.96 * 15)
#> [1] 70.6 129.4
```

Notice that we quote the binary operator when defining it but not when using it.

The pleasure of defining your own operators is that you can wrap commonly used operations inside a succinct syntax. If your application frequently concatenates two strings without an intervening blank, then you might define a binary concatenation operator for that purpose:

```
'%+%' <- function(s1, s2)
  paste(s1, s2, sep = "")
"Hello" %+%" World"
#> [1] "HelloWorld"
"limit=" %+% round(qnorm(1 - 0.05 / 2), 2)
#> [1] "limit=1.96"
```

A danger of defining your own operators, however, is that the code becomes less portable to other environments. Bring the definitions along with the code in which they are used; otherwise, R will complain about undefined operators.

All user-defined operators have the same precedence and are listed collectively in [Table 2-1](#) as `%any%`. Their precedence is fairly high: higher than multiplication and division but lower than exponentiation and sequence creation. As a result, it's easy to misexpress yourself. If we omit parentheses from the `%+-%` example, we get an unexpected result:

```
100 %+-% 1.96 * 15
#> [1] 1471 1529
```

R interpreted the expression as $(100 \%+-\% 1.96) * 15$.

See Also

See [Recipe 2.11](#) for more about operator precedence and [Recipe 15.3](#) for how to define a function.

12.18 Suppressing the Startup Message

Problem

When you run R from a command prompt or shell script, you are tired of seeing R's verbose startup message.

Solution

Use the `--quiet` command-line option when you start R from the command line or a shell script.

Discussion

The startup message from R is handy for beginners because it contains useful information about the R project and getting help. But the novelty wears off pretty quickly—especially if you start R from a shell prompt to use it as a calculator throughout the day. This is not particularly helpful if you’re using R only from RStudio.

If you start R from the shell prompt, use the `--quiet` option to hide the startup message:

```
R --quiet
```

On a Linux or Mac box, you could alias R like this from the shell so you never see the startup message:

```
alias R="/usr/bin/R --quiet"
```

12.19 Getting and Setting Environment Variables

Problem

You want to see the value of an environment variable, or you want to change its value.

Solution

Use the `Sys.getenv` function to see values. Use `Sys.putenv` to change them:

```
Sys.setenv(DB_PASSWORD = "My_Password!")
Sys.getenv("DB_PASSWORD")
#> [1] "My_Password!"
```

Discussion

Environment variables are often used to configure and control software. Each process has its own set of environment variables, which are inherited from its parent process. You sometimes need to see the environment variable settings for your R process in order to understand its behavior. Likewise, you sometimes need to change those settings to modify that behavior.

A common use case is to store a username or password for use in accessing a remote database or cloud service. It’s a really bad idea to store passwords in plain text in a project script. One way to avoid storing passwords in your script is to set an environment variable containing your password when R starts.

To ensure your password and username are available at every R login, you can add calls to `Sys.setenv` in the `.Rprofile` file in your home directory. `.Rprofile` is an R script that is run every time R starts.

For example, you could add the following to your *.Rprofile*:

```
Sys.setenv(DB_USERID = "Me")
Sys.setenv(DB_PASSWORD = "My_Password!")
```

Then you could fetch and use the environment variables in a script to log into an Amazon Redshift database, for example:

```
con <- DBI::dbConnect(
  RPostgreSQL::PostgreSQL(),
  dbname   = "my_database",
  port     = 5439,
  host     = "my_database.amazonaws.com",
  user     = Sys.getenv("DB_USERID"),
  password = Sys.getenv("DB_PASSWORD"))
)
```

See Also

See [Recipe 3.16](#) for more about changing configuration at startup.

12.20 Use Code Sections

Problem

You've got a long script and you're finding it difficult to navigate from one section of code to the next.

Solution

Code sections provide section dividers in an outline pane on the side of your editor. To use code sections, simply start a comment with # and then end the comment with ---- or ##### or =====:

```
# My First Section      -----
x <- 1

# My Second Section    #####
y <- 2

# My Third Section     =====
z <- 3
```

In the RStudio editor window you can see the outline on the righthand side (see [Figure 12-2](#)).

A screenshot of the RStudio interface. The left pane shows R code with three sections: "My First Section", "My Second Section", and "My Third Section". The right pane shows an outline with the same three sections. The top bar has various icons and buttons, including "Source on Save", "Run", and "Source".

```
1
2 # My First Section      -----
3 x <- 1
4
5 # My Second Section   #####
6 y <- 2
7
8 # My Third Section    -----
9 z <- 3
10
```

My First Section
My Second Section
My Third Section

Figure 12-2. Code sections

Discussion

Code sections are just a specially formatted type of R comment since they start with the `#` symbol. If you open your code with any editor other than RStudio, they are treated simply as code comments. But RStudio sees these specially formatted code comments as section headers and creates a helpful outline in the side panel of the editor.



The first time you use code sections, you may need to click the outline icon to the right of the Source button in order to show the outline.

If you are writing R Markdown instead of a `*.R` script, your Markdown headings and subheadings will show up in the outline pane, making navigating your document much easier.

See Also

See [Recipe 16.4](#) for using section headings in R Markdown documents.

12.21 Executing R in Parallel Locally

Problem

You have code that takes a while to run, and you would like to speed it up by using more of the cores on your local computer.

Solution

The easiest solution to get up and running with is to use the `furrr` package, which in turn uses the `future` package to provide parallel processing via functions that feel like those from `purrr` except that they operate in parallel.

You'll want to download the latest development version from GitHub because the package is still under active development as of this writing:

```
devtools::install_github("DavisVaughan/furrr")
```

To use `furrr` to parallelize our code, we call the `furrr::future_map` function in place of the `purrr::map` function we discussed in [Recipe 6.1](#). But first we have to tell `furrr` how we want to parallelize. In this case we want a `multiprocess` parallel process that uses all our local processors, so we set that up by calling `plan(multiprocess)`. Then we can apply a function to every element in our list using `future_map`:

```
library(furrr)  
  
plan(multiprocess)  
  
future_map(my_list, some_function)
```

Discussion

Let's do an example simulation to illustrate parallelization. A classic stochastic simulation is to draw random points inside of a 2×2 box and see how many points fall within one unit from the center of the box. The ratio of points inside the box / total points multiplied by 4 is a good estimate of pi. The following function takes one input, `n_iterations`, which is the number of random points to simulate. Then it returns the resulting average estimate of pi:

```
simulate_pi <- function(n_iterations) {  
  rand_draws <- matrix(runif(2 * n_iterations, -1, 1), ncol = 2)  
  num_in <- sum(sqrt(rand_draws[, 1]**2 + rand_draws[, 2]**2) <= 1)  
  pi_hat <- (num_in / n_iterations) * 4  
  return(pi_hat)  
}  
simulate_pi(1000000)  
#> [1] 3.14
```

As you can see, even with 1,000,000 simulations the result is only accurate out to a couple of decimal points. This is not a very efficient way to estimate pi, but it works for our illustration.

For the purpose of comparison later, let's run 200 runs of this pi simulator where each run has 2,500,000 simulated points. We'll do this by creating a list with 200 elements, each of which is the value 5,000,000, which we will pass to `simulate_pi`. We'll time the code with the `tictoc` package:

```

library(purrr) # for `map`
library(tictoc) # for timing our code

draw_list <- as.list(rep(5000000, 200))

tic("simulate pi - single process")
sims_list <- map(draw_list, simulate_pi)
toc()
#> simulate pi - single process: 90.772 sec elapsed

mean(unlist(sims_list))
#> [1] 3.14

```

That runs in less than two minutes and gives an estimate of pi based on a billion simulations ($5m \times 200$).

Now let's take the exact same R function, `simulate_pi`, and run it through `future_map` to run it in parallel:

```

library(furrr)
#> Loading required package: future
#>
#> Attaching package: 'future'
#> The following object is masked from 'package:tseries':
#>
#>     value
plan(multiprocess)

tic("simulate pi - parallel")
sims_list <- future_map(draw_list, simulate_pi)
toc()
#> simulate pi - parallel: 26.33 sec elapsed
mean(unlist(sims_list))
#> [1] 3.14

```

The preceding example was run on a MacBook Pro with four physical cores and two virtual cores per physical core. When you're running code in parallel the best-case scenario is that the runtime is reduced by $1/(\text{number of physical cores})$. With four physical cores you can see the parallel runtime is much faster than the single-threaded version, but not quite one-fourth the runtime of the single-threaded version. There is always some overhead from moving the data around, so you will never experience the best-case scenario. And the more data each iteration produces, the less speed improvement you will experience from parallelization.

See Also

See [Recipe 12.22](#).

12.22 Executing R in Parallel Remotely

Problem

You have access to a number of remote machines and you would like to run your code in parallel across them all.

Solution

Running code in parallel across multiple machines can be tricky to set up initially. However, if we start with a few key prerequisites in place, the process has a much higher probability of success.

The starting prerequisites are:

- You can ssh from your main machine to each remote node without a password using previously generated SSH keys.
- The remote nodes all have R installed (ideally the same version of R).
- Paths are set such that you can run Rscript from SSH.
- The remote nodes have the package `furrr` installed (which in turn installs `future`).
- The remote nodes already have all the packages your distributed code depends on installed.

Once you have worker nodes that are set up and ready to go, you can create a cluster by calling `makeClusterPSOCK` from the `future` package. Then use the resulting cluster with the `furrr` function `future_map`:

```
library(furrr) # loads future as a dependency

workers <- c("node_1.domain.com", "node_2.domain.com")

cl <- makeClusterPSOCK(
  worker = workers
)

plan(cluster, workers = cl)

future_map(my_list, some_function)
```

Discussion

Suppose we have two big Linux machines named `von-neumann12` and `von-neumann15` that we can use to run numerical models. These machines meet the criteria just listed,

so they are good candidates to be our backend for a `furrr`/future cluster. Let's do the same pi simulation we did in the previous recipe using the `simulate_pi` function:

```
library(tidyverse)
library(furrr)
library(tictoc)

my_workers <- c('von-neumann12', 'von-neumann15')

cl <- makeClusterPSOCK(
  workers = my_workers,
  rscript = '/home/anaconda2/bin/Rscript', #yours may differ
  verbose=TRUE
)

draw_list <- as.list(rep(5000000, 200))

plan(cluster, workers = cl)

tic('simulate pi - parallel map')
sims_list_parallel <- draw_list %>%
  future_map(simulate_pi)
toc()
#> simulate pi - parallel map: 116.986 sec elapsed

mean(unlist(sims_list_parallel))
#> [1] 3.14167
```

This is ~8.5 million sims per second.

The two nodes in our ad hoc cluster each have 32 processors and 128 GB of RAM. But if you compare the runtime of the preceding code with the runtime of the prior recipe run on a humble MacBook Pro, you'll notice that the MacBook executed the code in about the same time as the multi-CPU Linux cluster with 64 total processors! This unintuitive surprise happens because the preceding code runs only on one CPU per cluster node. So, as a result, it uses only two CPUs, while the MacBook uses all four of its CPUs.

So how do we run parallel code on a cluster and have each node also run in parallel across multiple CPU cores? To do that we need to make three changes to our code:

1. Create a nested parallel plan that uses `both cluster` and `multiprocess`.
2. Create an input list that is a nested list. Each cluster machine will get from the main list an item that contains sublist items that it can process in parallel across all its CPUs.
3. Call `future_map` twice, using a nested call. The outer `future_map` will parallelize items across the cluster nodes, and then the inner call will parallelize across the CPUs.

To created the nested parallel plan, we will create a multipart plan by passing a list of two plans to the `plan` function like this:

```
plan(list(tweak(cluster, workers = cl), multiprocess))
```

The second change is to create the nested list to iterate on. We can do that by using the `split` command and passing it our prior list followed by a vector of `1:4`, like so:

```
split(draw_list, 1:4)
```

This will break the initial list into four sublists, so our resulting list will have four elements. Each sublist will have 50 inputs for our final `simulate_pi` function.

The third change to our code is to create a nested `future_map` call that will pass each of our four list elements to the worker nodes, which subsequently will iterate over the elements of each sublist. We create that nested function like this:

```
future_map(draw_list, ~future_map(., simulate_pi))
```

The `~` sets up R to expect an anonymous function inside the first `future_map` call, and the `.` tells R where to put the list element. The anonymous function in this example is a separate call to `future_map` that gets executed on each node.

Here are all three changes integrated into the code:

```
# nested parallel plan - the first part of the plan is the cluster call
# followed by the multiprocess
plan(list(tweak(cluster, workers = cl), multiprocess))

# break the draw_list into a nested list with fewer elements
draw_list_nested <- split(draw_list, 1:4)

tic('simulate pi - parallel nested map')
sims_list_nested_parallel <- future_map(
  draw_list_nested, ~future_map(., simulate_pi)
)
toc()
#> simulate pi - parallel nested map: 15.964 sec elapsed
mean(unlist(sims_list_nested_parallel))
#> [1] 3.14158
```

You can see the runtime decreased substantially from the previous example, although with 32 processors on each node, we're not seeing a 32 \times improvement in runtime. This is because we're passing only 50 sets of simulations to each node. Each node runs 32 sets of simulations in the first pass but only 18 in the second pass, leaving half the CPUs idle.

Let's keep the CPUs a little busier by increasing our total simulations from 1 billion to 25 billion. Then we'll break them into 500 work blocks to be spread to the two worker nodes:

```
draw_list <- as.list(rep(5000000, 5000))
draw_list_nested <- split(draw_list, 1:50)

plan(list(tweak(cluster, workers = cl), multiprocess))

tic('simulate pi - parallel nested map')
sims_list_nested_parallel <- future_map(
  draw_list_nested, ~future_map(., simulate_pi)
)
toc()
#> simulate pi - parallel nested map: 260.532 sec elapsed
mean(unlist(sims_list_nested_parallel))
#> [1] 3.14157
```

This gives us \sim 96 million sims per second.

See Also

The `future` package has multiple excellent vignettes. To better understand the nested `plan` call, start with `vignette('future-3-topologies', package = 'future')`.

Further info about `furr` can be found at its [GitHub page](#).

Beyond Basic Numerics and Statistics

This chapter presents a few advanced techniques such as those you might encounter in the first or second year of a graduate program in applied statistics.

Most of these recipes use functions available in the base distribution. Through add-on packages, R provides some of the world's most advanced statistical techniques. This is because researchers in statistics now use R as their *lingua franca*, showcasing their newest work. Anyone looking for a cutting-edge statistical technique is urged to search CRAN and the web for possible implementations.

13.1 Minimizing or Maximizing a Single-Parameter Function

Problem

Given a single-parameter function f , you want to find the point at which f reaches its minimum or maximum.

Solution

To minimize a single-parameter function, use `optimize`. Specify the function to be minimized and the bounds for its domain (x):

```
optimize(f, lower = lowerBound, upper = upperBound)
```

If you instead want to maximize the function, specify `maximum = TRUE`:

```
optimize(f,
         lower = lowerBound,
         upper = upperBound,
         maximum = TRUE)
```

Discussion

The `optimize` function can handle functions of one argument. It requires upper and lower bounds for `x` that delimit the region to be searched. The following example finds the minimum of a polynomial, $3x^4 - 2x^3 + 3x^2 - 4x + 5$:

```
f <- function(x)
  3 * x ^ 4 - 2 * x ^ 3 + 3 * x ^ 2 - 4 * x + 5
optimize(f, lower = -20, upper = 20)
#> $minimum
#> [1] 0.597
#>
#> $objective
#> [1] 3.64
```

The returned value is a list with two elements: `minimum`, the `x` value that minimizes the function; and `objective`, the value of the function at that point.

A tighter range for `lower` and `upper` means a smaller region to be searched and hence a faster optimization. However, if you are unsure of the appropriate bounds, use big but reasonable values such as `lower = -1000` and `upper = 1000`. Just be careful that your function does not have multiple minima within that range! The `optimize` function will find and return only one such minimum.

See Also

See [Recipe 13.2](#).

13.2 Minimizing or Maximizing a Multiparameter Function

Problem

Given a multiparameter function `f`, you want to find the point at which `f` reaches its minimum or maximum.

Solution

To minimize a multiparameter function, use `optim`. You must specify the starting point, which is a vector of initial arguments for `f`:

```
optim(startingPoint, f)
```

To maximize the function instead, specify this control parameter:

```
optim(startingPoint, f, control = list(fnscale = -1))
```

Discussion

The `optim` function is more general than `optimize` (see [Recipe 13.1](#)) because it handles multiparameter functions. To evaluate your function at a point, `optim` packs the point's coordinates into a vector and calls your function with that vector. The function should return a scalar value. `optim` will begin at your starting point and move through the parameter space, searching for the function's minimum.

Here is an example of using `optim` to fit a nonlinear model. Suppose you believe that the paired observations z and x are related by $z_i = (x_i + \alpha)^\beta + \varepsilon_i$, where α and β are unknown parameters and where the ε_i are nonnormal noise terms. Let's fit the model by minimizing a robust metric, the sum of the absolute deviations:

$$\sum |z - (x + a)^b|$$

First we define the function to be minimized. Note that the function has only one formal parameter, a two-element vector. The actual parameters to be evaluated, a and b , are packed into the vector in locations 1 and 2:

```
load(file = './data/opt.rdata') # loads x, y, z

f <-
  function(v) {
    a <- v[1]
    b <- v[2]                                # "unpack" v, giving a and b
    sum(abs(z - ((x + a) ^ b)))            # calculate and return the error
  }
```

The following code makes a call to `optim`, starts from $(1, 1)$, and searches for the minimum point of `f`:

```
optim(c(1, 1), f)
#> $par
#> [1] 10.0  0.7
#>
#> $value
#> [1] 1.26
#>
#> $counts
#> function gradient
#>      485      NA
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

The returned list includes `convergence`, the indicator of success or failure. If this indicator is 0, then `optim` found a minimum; otherwise, it did not. Obviously, the

convergence indicator is the most important returned value because other values are meaningless if the algorithm did not converge.

The returned list also includes `par`, the parameters that minimize our function, and `value`, the value of `f` at that point. In this case, `optim` did converge and found a minimum point at approximately $a = 10.0$ and $b = 0.7$.



There are no lower and upper bounds for `optim`, just the starting point that you provide. A better guess for the starting point means a faster minimization.

The `optim` function supports several different minimization algorithms, and you can select among them. If the default algorithm does not work for you, see the help page for alternatives. A typical problem with multidimensional minimization is that the algorithm gets stuck at a local minimum and fails to find a deeper, global minimum. Generally speaking, the algorithms that are more powerful are less likely to get stuck. However, there is a trade-off: they also tend to run more slowly.

See Also

The R community has implemented many tools for optimization. On CRAN, see the task view for [Optimization and Mathematical Programming](#) for more solutions.

13.3 Calculating Eigenvalues and Eigenvectors

Problem

You want to calculate the eigenvalues or eigenvectors of a matrix.

Solution

Use the `eigen` function. It returns a list with two elements, `values` and `vectors`, which contain (respectively) the eigenvalues and eigenvectors.

Discussion

Suppose we have a matrix such as the Fibonacci matrix:

```
fibmat <- matrix(c(0, 1, 1, 1), 2, 2)
fibmat
#>      [,1] [,2]
#> [1,]     0     1
#> [2,]     1     1
```

Given the matrix, the `eigen` function will return a list of its eigenvalues and eigenvectors:

```
eigen(fibmat)
#> eigen() decomposition
#> $values
#> [1] 1.618 -0.618
#>
#> $vectors
#>      [,1]   [,2]
#> [1,] 0.526 -0.851
#> [2,] 0.851  0.526
```

Use either `eigen(fibmat)$values` or `eigen(fibmat)$vectors` to select the needed value from the list.

13.4 Performing Principal Component Analysis

Problem

You want to identify the principal components of a multivariable dataset.

Solution

Use the `prcomp` function. The first argument is a formula whose righthand side is the set of variables, separated by plus signs (+). The lefthand side is empty:

```
r <- prcomp(~ x + y + z)
summary(r)
#> Importance of components:
#>                 PC1     PC2     PC3
#> Standard deviation 1.894 0.11821 0.04459
#> Proportion of Variance 0.996 0.00388 0.00055
#> Cumulative Proportion 0.996 0.99945 1.00000
```

Discussion

Base R includes two functions for principal component analysis (PCA), `prcomp` and `princomp`. The documentation mentions that `prcomp` has better numerical properties, so that's the function presented here.

An important use of PCA is to reduce the dimensionality of your dataset. Suppose your data contains a large number N of variables. Ideally, all the variables are more or less independent and contributing equally. But if you suspect that some variables are redundant, PCA can tell you the number of sources of variance in your data. If that number is near N , then all the variables are useful. If the number is less than N , then your data can be reduced to a dataset of smaller dimensionality.

PCA recasts your data into a vector space where the first dimension captures the most variance, the second dimension captures the second most, and so forth. The actual output from `prcomp` is an object that, when printed, gives the needed vector rotation:

```
load(file = './data/pca.rdata')
r <- prcomp(~ x + y)
print(r)
#> Standard deviations (1, .., p=2):
#> [1] 0.393 0.163
#>
#> Rotation (n x k) = (2 x 2):
#>          PC1    PC2
#> x -0.553  0.833
#> y -0.833 -0.553
```

We typically find the summary of PCA much more useful. It shows the proportion of variance that is captured by each component:

```
summary(r)
#> Importance of components:
#>                      PC1    PC2
#> Standard deviation   0.393 0.163
#> Proportion of Variance 0.853 0.147
#> Cumulative Proportion  0.853 1.000
```

In this example, the first component captured 85% of the variance and the second component only 15%, so we know the first component captured most of it.

After calling `prcomp`, use `plot(r)` to view a bar chart of the variances of the principal components and `predict(r)` to rotate your data to the principal components.

See Also

See [Recipe 13.9](#) for an example of using principal component analysis. Further uses of PCA in R are discussed in *Modern Applied Statistics with S-Plus* by W. N. Venables and B. D. Ripley (Springer).

13.5 Performing Simple Orthogonal Regression

Problem

You want to create a linear model using orthogonal regression in which the variances of x and y are treated symmetrically.

Solution

Use `prcomp` to perform PCA on x and y . From the resulting rotation, compute the slope and intercept:

```
r <- prcomp(~ x + y)
slope <- r$rotation[2, 1] / r$rotation[1, 1]
intercept <- r$center[2] - slope * r$center[1]
```

Discussion

Orthogonal regression is also known as *total least squares* (TLS).

The ordinary least squares (OLS) algorithm has an odd property: it is asymmetric. That is, calculating $\text{lm}(y \sim x)$ is not the mathematical inverse of calculating $\text{lm}(x \sim y)$. The reason is that OLS assumes the x values to be constants and the y values to be random variables, so all the variance is attributed to y and none is attributed to x . This creates an asymmetric situation.

The asymmetry is illustrated in [Figure 13-1](#), where the upper-left panel displays the fit for $\text{lm}(y \sim x)$. The OLS algorithm tries to minimize the vertical distances, which are shown as dotted lines. The upper-right panel shows the identical dataset but fit with $\text{lm}(x \sim y)$ instead, so the algorithm is minimizing the horizontal dotted lines. Obviously, you'll get a different result depending upon which distances are minimized.

The lower panel of [Figure 13-1](#) is quite different. It uses PCA to implement orthogonal regression. Now the distances being minimized are the orthogonal distances from the data points to the regression line. This is a symmetric situation: reversing the roles of x and y does not change the distances to be minimized.

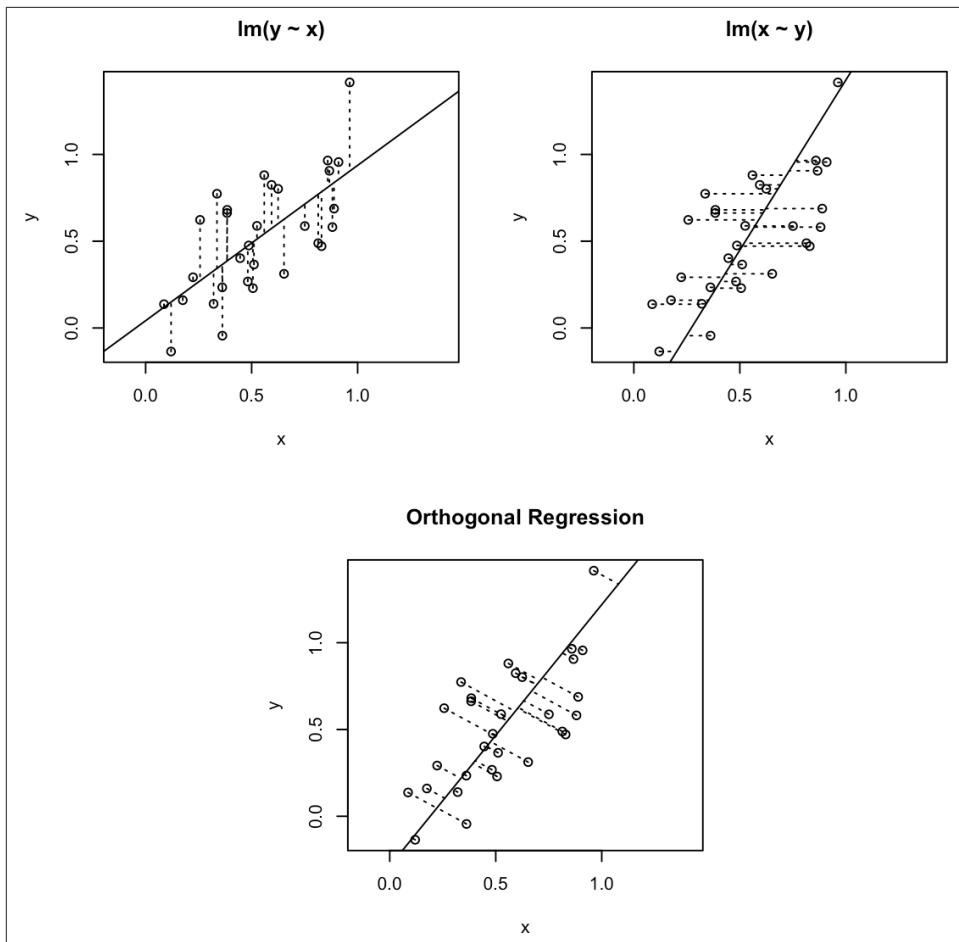


Figure 13-1. Ordinary least squares versus orthogonal regression

Implementing a basic orthogonal regression in R is quite simple. First, perform the PCA:

```
load(file = './data/pca.rdata')
r <- prcomp(~ x + y)
```

Next, use the rotations to compute the slope:

```
slope <- r$rotation[2, 1] / r$rotation[1, 1]
```

And then, from the slope, calculate the intercept:

```
intercept <- r$center[2] - slope * r$center[1]
```

We call this a “basic” regression because it yields only the point estimates for the slope and intercept, not the confidence intervals. Obviously, we’d like to have the regression

statistics, too. [Recipe 13.8](#) shows one way to estimate the confidence intervals using a bootstrap algorithm.

See Also

Principal component analysis is described in [Recipe 13.4](#). The graphics in this recipe were inspired by the work of Vincent Zoonekynd and his [tutorial on regression](#).

13.6 Finding Clusters in Your Data

Problem

You believe your data contains clusters: groups of points that are “near” each other. You want to identify those clusters.

Solution

Your dataset, `x`, can be a vector, data frame, or matrix. Assume that n is the number of clusters you desire:

```
d <- dist(x)          # Compute distances between observations
hc <- hclust(d)        # Form hierarchical clusters
clust <- cutree(hc, k=n) # Organize them into the n largest clusters
```

The result, `clust`, is a vector of numbers between 1 and n , one for each observation in `x`. Each number classifies its corresponding observation into one of the n clusters.

Discussion

The `dist` function computes distances between all the observations. The default is Euclidean distance, which works well for many applications, but other distance measures are also available.

The `hclust` function uses those distances to form the observations into a hierarchical tree of clusters. You can plot the result of `hclust` to create a visualization of the hierarchy, called a *dendrogram*, as shown in [Figure 13-2](#).

Finally, `cutree` extracts clusters from that tree. You must specify either how many clusters you want or the height at which the tree should be cut. Often the number of clusters is unknown, in which case you will need to explore the dataset for clustering that makes sense in your application.

We’ll illustrate clustering of a synthetic dataset. We start by generating 99 normal variates, each with a randomly selected mean of either -3, 0, or +3:

```
means <- sample(c(-3, 0, +3), 99, replace = TRUE)
x <- rnorm(99, mean = means)
```

For our own curiosity, we can compute the true means of the original clusters. (In a real situation, we would not have the `means` factor and would be unable to perform this computation.) We can confirm that the groups' means are pretty close to -3, 0, and +3:

```
tapply(x, factor(means), mean)
#>      -3       0       3
#> -3.015 -0.224  2.760
```

To “discover” the clusters, we first compute the distances between all points:

```
d <- dist(x)
```

Then we create the hierarchical clusters:

```
hc <- hclust(d)
```

And we can plot the hierarchical cluster dendrogram by calling `plot` on the `hc` object (Figure 13-2):

```
plot(hc,
      sub = "",
      labels = FALSE)
```

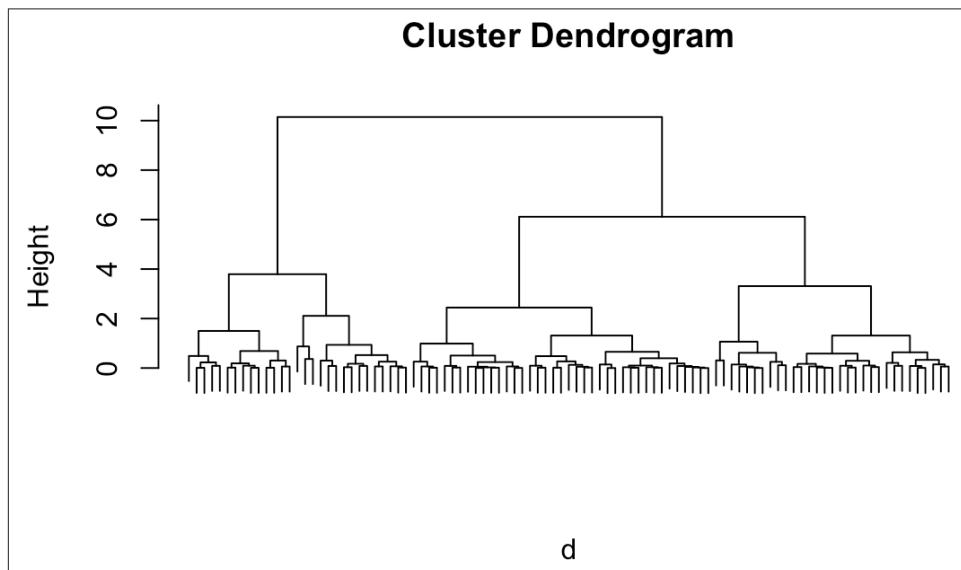


Figure 13-2. Hierarchical cluster dendrogram

We can now extract the three largest clusters:

```
clust <- cutree(hc, k=3)
```

Obviously, we have a huge advantage here because we know the true number of clusters. Real life is rarely that easy. However, even if we didn't already know we were dealing with three clusters, looking at the dendrogram gives us a good clue that there are three big clusters in the data.

`clust` is a vector of integers between 1 and 3, one integer for each observation in the sample, that assigns each observation to a cluster. Here are the first 20 cluster assignments:

```
head(clust, 20)
#> [1] 1 2 2 2 1 2 3 3 2 3 1 3 2 3 2 1 2 1 1 3
```

By treating the cluster number as a factor, we can compute the mean of each statistical cluster (see [Recipe 6.6](#)):

```
tapply(x, clust, mean)
#>      1      2      3
#>  3.190 -2.699  0.236
```

R did a good job of splitting the data into clusters: the means appear distinct, with one near -2.7 , one near 0.27 , and one near $+3.2$. (The order of the extracted means does not necessarily match the order of the original groups, of course.) The extracted means are similar but not identical to the original means. Side-by-side boxplots can show why (see [Figure 13-3](#)):

```
library(patchwork)

df_cluster <- data.frame(x,
                          means = factor(means),
                          clust = factor(clust))

g1 <- ggplot(df_cluster) +
  geom_boxplot(aes(means, x)) +
  labs(title = "Original Clusters", x = "Cluster Mean")

g2 <- ggplot(df_cluster) +
  geom_boxplot(aes(clust, x)) +
  labs(title = "Identified Clusters", x = "Cluster Number")

g1 + g2
```

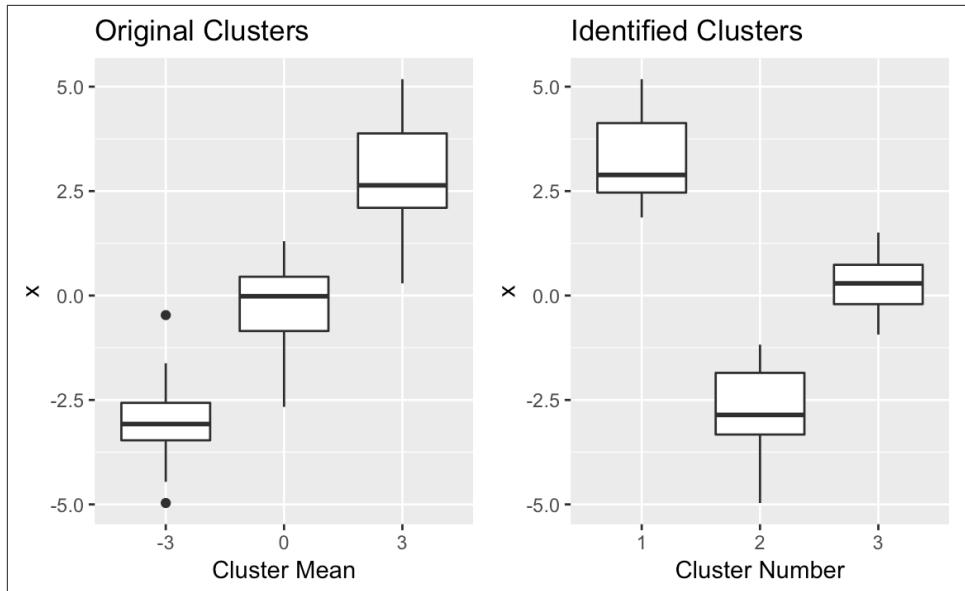


Figure 13-3. Cluster boxplots

The clustering algorithm perfectly separated the data into nonoverlapping groups. The original clusters overlapped, whereas the identified clusters do not.

This illustration used one-dimensional data, but the `dist` function works equally well on multidimensional data stored in a data frame or matrix. Each row in the data frame or matrix is treated as one observation in a multidimensional space, and `dist` computes the distances between those observations.

See Also

This demonstration is based on the clustering features of the base package. There are other packages, such as `mclust`, that offer alternative clustering mechanisms.

13.7 Predicting a Binary-Valued Variable (Logistic Regression)

Problem

You want to perform logistic regression, a regression model that predicts the probability of a binary event occurring.

Solution

Call the `glm` function with `family = binomial` to perform logistic regression. The result is a model object:

```
m <- glm(b ~ x1 + x2 + x3, family = binomial)
```

Here, `b` is a factor with two levels (e.g., `TRUE` and `FALSE`, 0 and 1), while `x1`, `x2`, and `x3` are predictor variables.

Use the model object, `m`, and the `predict` function to predict a probability from new data:

```
df <- data.frame(x1 = value, x2 = value, x3 = value)
predict(m, type = "response", newdata = df)
```

Discussion

Predicting a binary-valued outcome is a common problem in modeling. Will a treatment be effective or not? Will prices rise or fall? Who will win the game, team A or team B? Logistic regression is useful for modeling these situations. In the true spirit of statistics, it does not simply give a “thumbs up” or “thumbs down” answer; rather, it computes a probability for each of the two possible outcomes.

In the call to `predict`, we set `type = "response"` so that `predict` returns a probability. Otherwise, it returns log-odds, which most of us don't find intuitive.

In his unpublished book entitled *Practical Regression and ANOVA Using R*, Julian Faraway gives an example of predicting a binary-valued variable: `test` from the dataset `pima` is true if the patient tested positive for diabetes. The predictors are diastolic blood pressure and body mass index (BMI). Faraway uses linear regression, so let's try logistic regression instead:

```
data(pima, package = "faraway")
b <- factor(pima$test)
m <- glm(b ~ diastolic + bmi, family = binomial, data = pima)
```

The summary of the resulting model, `m`, shows that the respective *p*-values for the `diastolic` and `bmi` variables are 0.8 and (essentially) 0. We can therefore conclude that only the `bmi` variable is significant:

```
summary(m)
#>
#> Call:
#> glm(formula = b ~ diastolic + bmi, family = binomial, data = pima)
#>
#> Deviance Residuals:
#>    Min      1Q  Median      3Q     Max
#> -1.913  -0.918  -0.685   1.234   2.742
#>
```

```

#> Coefficients:
#>             Estimate Std. Error z value Pr(>|z|)
#> (Intercept) -3.62955   0.46818  -7.75 9.0e-15 ***
#> diastolic   -0.00110   0.00443  -0.25     0.8
#> bmi         0.09413   0.01230   7.65 1.9e-14 ***
#> ...
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for binomial family taken to be 1)
#>
#> Null deviance: 993.48 on 767 degrees of freedom
#> Residual deviance: 920.65 on 765 degrees of freedom
#> AIC: 926.7
#>
#> Number of Fisher Scoring iterations: 4

```

Because only the `bmi` variable is significant, we can create a reduced model like this:

```
m.red <- glm(b ~ bmi, family = binomial, data = pima)
```

Let's use the model to calculate the probability that someone with an average BMI (32.0) will test positive for diabetes:

```

newdata <- data.frame(bmi = 32.0)
predict(m.red, type = "response", newdata = newdata)
#>      1
#> 0.333

```

According to this model, the probability is about 33.3%. The same calculation for someone in the 90th percentile gives a probability of 54.9%:

```

newdata <- data.frame(bmi = quantile(pima$bmi, .90))
predict(m.red, type = "response", newdata = newdata)
#> 90%
#> 0.549

```

See Also

Using logistic regression involves interpreting the deviance to judge the significance of the model. We suggest you review a text on logistic regression before attempting to draw any conclusions from your regression.

13.8 Bootstrapping a Statistic

Problem

You have a dataset and a function to calculate a statistic from that dataset. You want to estimate a confidence interval for the statistic.

Solution

Use the `boot` package. Apply the `boot` function to calculate bootstrap replicates of the statistic:

```
library(boot)
bootfun <- function(data, indices) {
  # . . . calculate statistic using data[indices]. . .
  return(statistic)
}

reps <- boot(data, bootfun, R = 999)
```

Here, `data` is your original dataset, which can be stored in either a vector or a data frame. The statistic function (`bootfun` in this case) should expect two arguments: `data` and `indices`, a vector of integers that selects the bootstrap sample from `data`.

Next, use the `boot.ci` function to estimate a confidence interval from the replications:

```
boot.ci(reps, type = c("perc", "bca"))
```

Discussion

Anybody can calculate a statistic, but that's just the point estimate. We want to take it to the next level: what is the confidence interval (CI)? For some statistics, we can calculate the CI analytically. The CI for a mean, for instance, is calculated by the `t.test` function. Unfortunately, that is the exception and not the rule. For most statistics, the mathematics are too tortuous or simply unknown, and there is no known closed-form calculation for the CI.

The bootstrap algorithm can estimate a CI even when no closed-form calculation is available. It works like this. The algorithm assumes that you have a sample of size N and a function to calculate the statistic and performs the following steps:

1. Randomly select N elements from the sample, *sampling with replacement*. That set of elements is called a *bootstrap sample*.
2. Apply the function to the bootstrap sample to calculate the statistic. That value is called a *bootstrap replication*.
3. Repeat steps 1 and 2 many times to yield many (typically thousands) of bootstrap replications.
4. From the bootstrap replications, compute the confidence interval.

That last step may seem mysterious, but there are several algorithms for computing the CI. A simple one uses percentiles of the replications, such as taking the 2.5 percentile and the 97.5 percentile to form the 95% CI.

We're huge fans of the bootstrap because we work daily with obscure statistics, it is important that we know their confidence intervals, and there is definitely no known formula for obtaining those. The bootstrap gives us a good approximation.

Let's work an example. In [Recipe 13.4](#) we estimated the slope of a line using orthogonal regression. That gave us a point estimate, but how can we find the CI? First, we encapsulate the slope calculation within a function:

```
stat <- function(data, indices) {  
  r <- prcomp(~ x + y, data = data, subset = indices)  
  slope <- r$rotation[2, 1] / r$rotation[1, 1]  
  return(slope)  
}
```

Notice that the function is careful to select the subset defined by `indices` and to compute the slope from that exact subset.

Next, we calculate 999 replications of the slope. Recall that we had two vectors, x and y , in the original recipe; here, we combine them into a data frame:

```
load(file = './data/pca.rdata')  
library(boot)  
set.seed(3) # for reproducability  
  
boot.data <- data.frame(x = x, y = y)  
reps <- boot(boot.data, stat, R = 999)
```

The choice of 999 replications is a good starting point. You can always repeat the bootstrap with more and see if the results change significantly.

The `boot.ci` function can estimate the CI from the replications. It implements several different algorithms, and the `type` argument selects which algorithms are performed. For each selected algorithm, `boot.ci` will return the resulting estimate:

```
boot.ci(reps, type = c("perc", "bca"))  
#> BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS  
#> Based on 999 bootstrap replicates  
#>  
#> CALL :  
#> boot.ci(boot.out = reps, type = c("perc", "bca"))  
#>  
#> Intervals :  
#> Level      Percentile          BCa  
#> 95%    ( 1.07,  1.99 )   ( 1.09,  2.05 )  
#> Calculations and Intervals on Original Scale
```

Here we chose two algorithms, percentile and BCa, by setting `type = c("perc", "bca")`. The two resulting estimates appear at the bottom under their names. Other algorithms are available; see the help page for `boot.ci`.

You will note that the two confidence intervals are slightly different: (1.068, 1.992) versus (1.086, 2.050). This is an uncomfortable but inevitable result of using two different algorithms. We don't know any method for deciding which is better. If the selection is a critical issue, you will need to study the reference and understand the differences. In the meantime, our best advice is to be conservative and use the minimum lower bound and the maximum upper bound; in this case, that would be (1.068, 2.050).

By default, `boot.ci` estimates a 95% CI. You can change that via the `conf` argument, like this:

```
boot.ci(reps, type = c("perc", "bca"), conf = 0.90)
```

See Also

See [Recipe 13.4](#) for the slope calculation. A good tutorial and reference for the bootstrap algorithm is *An Introduction to the Bootstrap* by Bradley Efron and Robert Tibshirani (Chapman & Hall/CRC).

13.9 Factor Analysis

Problem

You want to perform factor analysis on your dataset, usually to discover what your variables have in common.

Solution

Use the `factanal` function, which requires your dataset and your estimate of the number of factors:

```
factanal(data, factors = n)
```

The output includes n factors, showing the loadings of each input variable for each factor.

The output also includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the number of factors is too small and does not capture the full dimensionality of the dataset; a p -value exceeding 0.05 indicates that there are likely enough (or more than enough) factors.

Discussion

Factor analysis creates linear combinations of your variables, called factors, that abstract the variables' underlying commonality. If your n variables are perfectly independent, then they have nothing in common and n factors are required to describe

them. But to the extent that the variables have an underlying commonality, fewer factors capture most of the variance and so fewer than n factors are required.

For each factor and variable, we calculate the correlation between them, known as the *loading*. Variables with a high loading are well explained by the factor. We can square the loading to know what fraction of the variable's total variance is explained by the factor.

Factor analysis is useful when it shows that a few factors capture most of the variance of your variables. Thus, it alerts you to redundancy in your data. In that case you can reduce your dataset by combining closely related variables or by eliminating redundant variables altogether.

A more subtle application of factor analysis is interpreting the factors to find interrelationships between your variables. If two variables both have large loadings for the same factor, then you know they have something in common. What is it? There is no mechanical answer. You'll need to study the data and its meaning.

There are two tricky aspects of factor analysis. The first is choosing the number of factors. Fortunately, you can use PCA to get a good initial estimate of the number of factors. The second tricky aspect is interpreting the factors themselves.

Let's illustrate factor analysis by using stock prices, or, more precisely, changes in stock prices. The dataset contains six months of price changes for the stocks of 12 companies. Every company is involved in the petroleum and gasoline industry. Their stock prices probably move together, since they are subject to similar economic and market forces. We might ask: how many factors are required to explain their changes? If only one factor is required, then all the stocks are the same and one is as good as another. If many factors are required, we know that owning several of them provides diversification.

We start by doing a PCA on `diffs`, the data frame of price changes. Plotting the PCA results shows the variance captured by the components ([Figure 13-4](#)):

```
load(file = './data/diffs.rdata')
plot(prcomp(diffs))
```

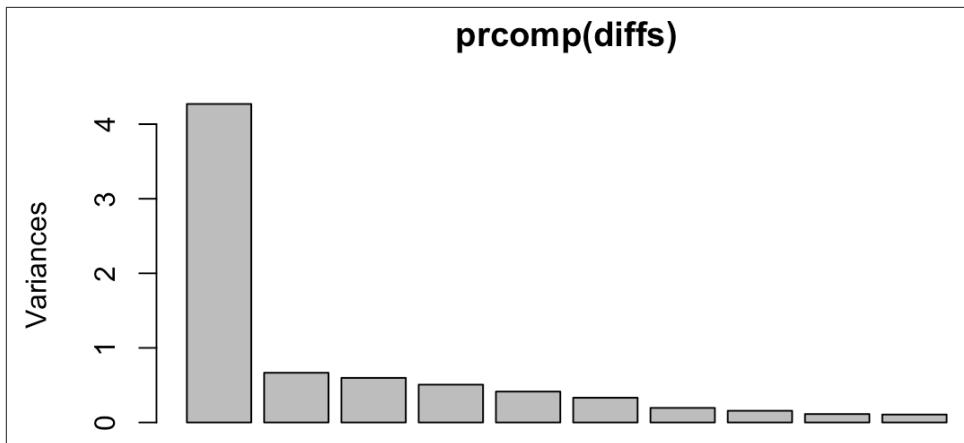


Figure 13-4. PCA results plot

We can see in Figure 13-4 that the first component captures much of the variance, but we don't know if more components are required. So we perform the initial factor analysis while assuming that two factors are required:

```
factanal(diffs, factors = 2)
#>
#> Call:
#> factanal(x = diffs, factors = 2)
#>
#> Uniquenesses:
#>   APC     BP    BRY    CVX    HES    MRO    NBL    OXY    ETP    VLO    XOM
#>   0.307  0.652  0.997  0.308  0.440  0.358  0.363  0.556  0.902  0.786  0.285
#>
#> Loadings:
#>   Factor1 Factor2
#> APC  0.773  0.309
#> BP   0.317  0.497
#> BRY
#> CVX  0.439  0.707
#> HES  0.640  0.389
#> MRO  0.707  0.377
#> NBL  0.749  0.276
#> OXY  0.562  0.358
#> ETP  0.283  0.134
#> VLO  0.303  0.350
#> XOM  0.355  0.767
#>
#>           Factor1 Factor2
#> SS loadings     2.98   2.072
#> Proportion Var  0.27   0.188
#> Cumulative Var 0.27   0.459
#>
#> Test of the hypothesis that 2 factors are sufficient.
```

```
#> The chi square statistic is 62.9 on 34 degrees of freedom.
#> The p-value is 0.00184
```

We can ignore most of the output because the *p*-value at the bottom is very close to zero (.00184). The small *p*-value indicates that two factors are insufficient, so the analysis isn't good. More are required, so we try again with three factors instead:

```
factanal(diffs, factors = 3)
#>
#> Call:
#> factanal(x = diffs, factors = 3)
#>
#> Uniquenesses:
#> APC     BP    BRY    CVX    HES    MRO    NBL    OXY    ETP    VLO    XOM
#> 0.316  0.650 0.984 0.315 0.374 0.355 0.346 0.521 0.723 0.605 0.271
#>
#> Loadings:
#>          Factor1 Factor2 Factor3
#> APC      0.747   0.270   0.230
#> BP       0.298   0.459   0.224
#> BRY      0.123
#> CVX      0.442   0.672   0.197
#> HES      0.589   0.299   0.434
#> MRO      0.703   0.350   0.167
#> NBL      0.760   0.249   0.124
#> OXY      0.592   0.357
#> ETP      0.194
#> VLO      0.198   0.264   0.535
#> XOM      0.355   0.753   0.190
#>
#>          Factor1 Factor2 Factor3
#> SS loadings    2.814   1.774   0.951
#> Proportion Var 0.256   0.161   0.086
#> Cumulative Var 0.256   0.417   0.504
#>
#> Test of the hypothesis that 3 factors are sufficient.
#> The chi square statistic is 30.2 on 25 degrees of freedom.
#> The p-value is 0.218
```

The large *p*-value (0.218) confirms that three factors are sufficient, so we can use the analysis.

The output includes a table of explained variance, shown here:

	Factor1	Factor2	Factor3
SS loadings	2.814	1.774	0.951
Proportion Var	0.256	0.161	0.086
Cumulative Var	0.256	0.417	0.504

This table shows that the proportion of variance explained by each factor is 0.256, 0.161, and 0.086, respectively. Cumulatively, they explain 0.504 of the variance, which leaves $1 - 0.504 = 0.496$ unexplained.

Next we want to interpret the factors, which is more like voodoo than science. Let's look at the loadings, repeated here:

	Loadings:		
	Factor1	Factor2	Factor3
APC	0.747	0.270	0.230
BP	0.298	0.459	0.224
BRY		0.123	
CVX	0.442	0.672	0.197
HES	0.589	0.299	0.434
MRO	0.703	0.350	0.167
NBL	0.760	0.249	0.124
OXY	0.592	0.357	
ETP	0.194		0.489
VLO	0.198	0.264	0.535
XOM	0.355	0.753	0.190

Each row is labeled with the variable name (stock symbol): APC, BP, BRY, and so forth. The first factor has many large loadings, indicating that it explains the variance of many stocks. This is a common phenomenon in factor analysis. We are often looking at related variables, and the first factor captures their most basic relationship. In this example, we are dealing with stocks, and most stocks move together in concert with the broad market. That's probably captured by the first factor.

The second factor is more subtle. Notice that the loadings for CVX (0.67) and XOM (0.75) are the dominant ones, with BP not far behind (0.46), but all other stocks have noticeably smaller loadings. This indicates a connection between CVX, XOM, and BP. Perhaps they operate together in a common market (e.g., multinational energy) and so tend to move together.

The third factor also has three dominant loadings: VLO, ETP, and HES. These are somewhat smaller companies than the global giants we saw in the second factor. Possibly these three share similar markets or risks and so their stocks also tend to move together.

In summary, it seems there are three groups of stocks here:

- CVX, XOM, BP
- VLO, ETP, HES
- Everything else

Factor analysis is an art and a science. We suggest that you read a good book on multivariate analysis before employing it.

See Also

See [Recipe 13.4](#) for more about PCA.

Time Series Analysis

Time series analysis has become a hot topic with the rise of quantitative finance and automated trading of securities. Many of the facilities described in this chapter were invented by practitioners and researchers in finance, securities trading, and portfolio management.

Before you start any time series analysis in R, a key decision is your choice of data representation (object class). This is especially critical in an object-oriented language such as R, because the choice affects more than how the data is stored; it also dictates which functions (methods) will be available for loading, processing, analyzing, printing, and plotting your data. When many people start using R they simply store time series data in vectors. That seems natural. However, they quickly discover that none of the coolest analytics for time series analysis work with simple vectors. We've found when users switch to using an object class intended for time series data, the analysis gets easier, opening a gateway to valuable functions and analytics.

This chapter's first recipe recommends using the `zoo` or `xts` packages for representing time series data. They are quite general and should meet the needs of most users. Nearly every subsequent recipe assumes you are using one of those two representations.



The `xts` implementation is a superset of `zoo`, so `xts` can do everything that `zoo` can do. In this chapter, whenever a recipe works for a `zoo` object, you can safely assume (unless stated otherwise) that it also works for an `xts` object.

Other Representations

Other representations of time series data are available in the R universe, including:

- The `fts` package
- The `irts` class from the `tseries` package
- The `timeSeries` package
- The `ts` class in the base distribution
- The `tsibble` package, a tidyverse style package for time series

In fact, there is a whole toolkit, called `tsbox`, just for converting between representations.

Two representations deserve special mention.

ts (base distribution)

The base distribution of R includes a time series class called `ts`. We don't recommend this representation for general use because the implementation itself is too limited and restrictive.

However, the base distribution includes some important time series analytics that depend upon `ts`, such as the autocorrelation function (`acf`) and the cross-correlation function (`ccf`). To use those base functions on `xts` data, use the `to.ts` function to "downshift" your data into the `ts` representation before calling the function. For example, if `x` is an `xts` object, you can compute its autocorrelation like this:

```
acf(as.ts(x))
```

tsibble package

The `tsibble` package is a recent extension to the tidyverse, specifically designed for working with time series data within the tidyverse. We find it useful for *cross-sectional* data—that is, data for which the observations are grouped by date, and you want to perform analytics *within* dates more than *across* dates.

Date Versus Datetime

Every observation in a time series has an associated date or time. The object classes used in this chapter, `zoo` and `xts`, give you the choice of using either dates or datetimes for representing the data's time component. You would use dates to represent daily data, of course, and also for weekly, monthly, or even annual data; in these cases, the date gives the day on which the observation occurred. You would use datetimes for intraday data, where both the date and time of observation are needed.

In describing this chapter’s recipes, we found it pretty cumbersome to keep saying “date or datetime.” So, we simplified the prose by assuming that your data is daily and thus uses whole dates. Please bear in mind, of course, that you are free and able to use timestamps below the resolution of a calendar date.

See Also

R has many useful functions and packages for time series analysis. You’ll find pointers to them in the [task view for Time Series Analysis](#).

14.1 Representing Time Series Data

Problem

You want an R data structure that can represent time series data.

Solution

We recommend the `zoo` and `xts` packages. They define a data structure for time series, and they contain many useful functions for working with time series data. Create a `zoo` object this way, where `x` is a vector, matrix, or data frame, and `dt` is a vector of corresponding dates or datetimes:

```
library(zoo)
ts <- zoo(x, dt)
```

Create an `xts` object in this way:

```
library(xts)
ts <- xts(x, dt)
```

Convert between representations of the time series data by using `as.zoo` and `as.xts`:

```
as.zoo(ts)
    Converts ts to a zoo object

as.xts(ts)
    Converts ts to an xts object
```

Discussion

R has at least eight different implementations of data structures for representing time series. We haven’t tried them all, but we can say that `zoo` and `xts` are excellent packages for working with time series data and better than the others that we have tried.

These representations assume you have two vectors: a vector of observations (data) and a vector of dates or times of those observations. The `zoo` function combines them into a `zoo` object:

```
library(zoo)
#>
#> Attaching package: 'zoo'
#> The following objects are masked from 'package:base':
#>
#>   as.Date, as.Date.numeric
x <- c(3, 4, 1, 4, 8)
dt <- seq(as.Date("2018-01-01"), as.Date("2018-01-05"), by = "days")

ts <- zoo(x, dt)
print(ts)
#> 2018-01-01 2018-01-02 2018-01-03 2018-01-04 2018-01-05
#>         3         4         1         4         8
```

The `xts` function is similar, returning an `xts` object:

```
library(xts)
#>
#> Attaching package: 'xts'
#> The following objects are masked from 'package:dplyr':
#>
#>   first, last
ts <- xts(x, dt)
print(ts)
#>           [,1]
#> 2018-01-01    3
#> 2018-01-02    4
#> 2018-01-03    1
#> 2018-01-04    4
#> 2018-01-05    8
```

The data, `x`, should be numeric. The vector of dates or datetimes, `dt`, is called the *index*. Legal indices vary between the packages:

`zoo`

The index can be any ordered values, such as `Date` objects, `POSIXct` objects, integers, or even floating-point values.

`xts`

The index must be a supported date or time class. This includes `Date`, `POSIXct`, and `chron` objects. Those should be sufficient for most applications, but you can also use `yearmon`, `yearqtr`, and `dateTime` objects. The `xts` package is more restrictive than `zoo` because it implements powerful operations that require a time-based index.

The following example creates a `zoo` object that contains the price of IBM stock for the first five days of 2010; it uses `Date` objects for the index:

```
prices <- c(132.45, 130.85, 130.00, 129.55, 130.85)
dates <- as.Date(c(
  "2010-01-04", "2010-01-05", "2010-01-06",
  "2010-01-07", "2010-01-08"
))
ibm.daily <- zoo(prices, dates)
print(ibm.daily)
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>     132      131      130      130      131
```

In contrast, the next example captures the price of IBM stock at one-second intervals. It represents time by the number of hours past midnight starting at 9:30 a.m. (1 second = 0.00027778 hours, more or less):

```
prices <- c(131.18, 131.20, 131.17, 131.15, 131.17)
seconds <- c(9.5, 9.500278, 9.500556, 9.500833, 9.501111)
ibm.sec <- zoo(prices, seconds)
print(ibm.sec)
#> 10 10 10 10 10
#> 131 131 131 131 131
```

Those two examples used a single time series, where the data came from a vector. Both `zoo` and `xts` can also handle multiple, parallel time series. For this, capture the several time series in a matrix or data frame and then create a multivariate time series by calling the `zoo` (or `xts`) function:

```
ts <- zoo(df, dt) # OR: ts <- xts(dfrm, dt)
```

The second argument is a vector of dates (or datetimes) for each observation. There is only one vector of dates for all the time series; in other words, all observations in each row of the matrix or data frame must have the same date. See [Recipe 14.5](#) if your data has mismatched dates.

Once the data is captured inside a `zoo` or `xts` object, you can extract the pure data via `coredata`, which returns a simple vector (or matrix):

```
coredata(ibm.daily)
#> [1] 132 131 130 130 131
```

You can extract the date or time portion via `index`:

```
index(ibm.daily)
#> [1] "2010-01-04" "2010-01-05" "2010-01-06" "2010-01-07" "2010-01-08"
```

The `xts` package is very similar to `zoo`. It is optimized for speed, so is especially well suited for processing large volumes of data. It is also clever about converting to and from other time series representations.

One big advantage of capturing data inside a `zoo` or `xts` object is that special-purpose functions become available for printing, plotting, differencing, merging, periodic sampling, applying rolling functions, and other useful operations. There is even a function, `read.zoo`, dedicated to reading time series data from ASCII files.

Remember that the `xts` package can do everything that the `zoo` package can do, so everywhere that this chapter talks about `zoo` objects you can also use `xts` objects.

If you are a serious user of time series data, we strongly recommend studying the documentation of these packages in order to learn about the ways they can improve your life. They are rich packages with many useful features.

See Also

See CRAN for documentation on `zoo` and `xts`, including reference manuals, vignettes, and quick reference cards. If the packages are already installed on your computer, view their documentation using the `vignette` function:

```
vignette("zoo")
vignette("xts")
```

The `timeSeries` package is another good implementation of a time series object. It is part of the Rmetrics project for quantitative finance.

14.2 Plotting Time Series Data

Problem

You want to plot one or more time series.

Solution

Use `plot(x)`, which works for `zoo` objects and `xts` objects containing either single or multiple time series.

For a simple vector `v` of time series observations, you can use either `plot(v, type = "l")` or `plot.ts(v)`.

Discussion

The generic `plot` function has a version for `zoo` objects and `xts` objects. It can plot objects that contain a single time series or multiple time series. In the latter case, it can plot each series in a separate plot or together in one plot.

Suppose that `ibm.infl` is a `zoo` object that contains two time series. One shows the quoted price of IBM stock from January 2000 through December 2017, and the other

is that same price adjusted for inflation. If you plot the object, R will plot the two time series together in one plot, as shown in [Figure 14-1](#):

```
load(file = "./data/ibm.rdata")
library(xts)

main <- "IBM: Historical vs. Inflation-Adjusted"
lty <- c("dotted", "solid")

# Plot the xts object
plot(ibm.infl,
     lty = lty, main = main,
     legend.loc = "left"
)
```

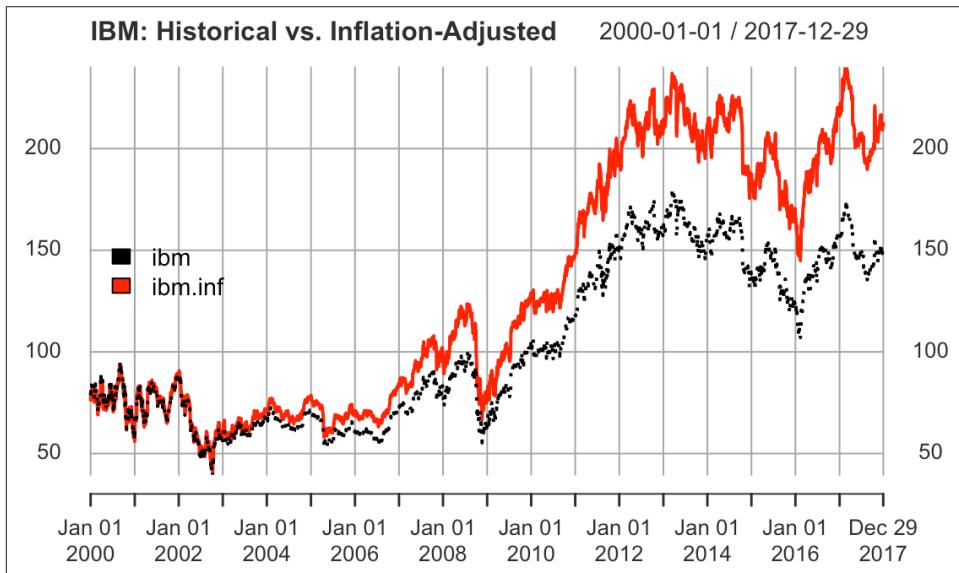


Figure 14-1. Example xts plot

The `plot` function for `xts` provides a default title as simply the name of the `xts` object. As we show here, it's common to set the `main` parameter to a more meaningful title.

The code specifies two line types (`lty`) so that the two lines are drawn in two different styles, making them easier to distinguish.

See Also

For working with financial data, the `quantmod` package contains special plotting functions that produce beautiful, stylized plots.

14.3 Extracting the Oldest or Newest Observations

Problem

You want to see only the oldest or newest observations of your time series.

Solution

Use `head` to view the oldest observations:

```
head(ts)
```

Use `tail` to view the newest observations:

```
tail(ts)
```

Discussion

The `head` and `tail` functions are generic, so they will work whether your data is stored in a simple vector, a `zoo` object, or an `xts` object.

Suppose you have an `xts` object with a multiyear history of the price of IBM stock, like the one used in the prior recipe. You can't display the whole dataset because it would scroll off your screen. But you can view the initial observations:

```
ibm <- ibm.infl$ibm # grab one column for illustration
head(ibm)
#>           ibm
#> 2000-01-01 78.6
#> 2000-01-03 82.0
#> 2000-01-04 79.2
#> 2000-01-05 82.0
#> 2000-01-06 80.6
#> 2000-01-07 80.2
```

And you can view the final observations:

```
tail(ibm)
#>           ibm
#> 2017-12-21 148
#> 2017-12-22 149
#> 2017-12-26 150
#> 2017-12-27 150
#> 2017-12-28 151
#> 2017-12-29 150
```

By default, `head` and `tail` show (respectively) the six oldest and six newest observations. You can see more observations by providing a second argument—for example, `tail(ibm, 20)`.

The `xts` package also includes `first` and `last` functions, which use calendar periods instead of number of observations. We can use `first` and `last` to select data by number of days, weeks, months, or even years:

```
first(ibm, "2 week")
#>           ibm
#> 2000-01-01 78.6
#> 2000-01-03 82.0
#> 2000-01-04 79.2
#> 2000-01-05 82.0
#> 2000-01-06 80.6
#> 2000-01-07 80.2
```

At first glance this output might be confusing. We asked for "2 week" and `xts` returned six days. That might seem off until we look at a calendar of January 2000 (Figure 14-2).

January 2000						
Su	Mo	Tu	We	Th	Fr	Sa
					1	
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Figure 14-2. January 2000 calendar

We can see from the calendar that the first week of January 2000 has only one day, Saturday the 1st. Then the second week runs from the 2nd to the 8th. Our data has no value for the 8th, so when we ask `first` for the first "2 week" it returns all the values from the first two calendar weeks. In our example dataset the first two calendar weeks contain only six values.

Similarly, we can ask `last` to give us the last month's worth of data:

```
last(ibm, "month")
#>           ibm
#> 2017-12-01 152
#> 2017-12-04 153
#> 2017-12-05 152
#> 2017-12-06 151
#> 2017-12-07 150
#> 2017-12-08 152
#> 2017-12-11 152
#> 2017-12-12 154
```

```
#> 2017-12-13 151  
#> 2017-12-14 151  
#> 2017-12-15 149  
#> 2017-12-18 150  
#> 2017-12-19 150  
#> 2017-12-20 150  
#> 2017-12-21 148  
#> 2017-12-22 149  
#> 2017-12-26 150  
#> 2017-12-27 150  
#> 2017-12-28 151  
#> 2017-12-29 150
```

If we had been using `zoo` objects here, we would need to have converted them to `xts` objects before passing the objects to `first` or `last`, as those are `xts` functions.

See Also

See `help(first.xts)` and `help(last.xts)` for details on the `first` and `last` functions, respectively.



The tidyverse package `dplyr` also has functions called `first` and `last`. If your workflow involves loading both the `xts` and `dplyr` packages, make sure to be explicit about which function you are calling by using the `package::function` notation (for example, `xts::first`).

14.4 Subsetting a Time Series

Problem

You want to select one or more elements from a time series.

Solution

You can index a `zoo` or `xts` object by position. Use one or two subscripts, depending upon whether the object contains one time series or multiple time series:

`ts[_i_]`

Selects the i th observation from a single time series

`ts[j, i]`

Selects the i th observation of the j th time series of multiple time series

You can index the time series by date. Use the same type of object as the index of your time series. This example assumes that the index contains `Date` objects:

```
ts[as.Date("yyyy-mm-dd")]
```

You can index it by a sequence of dates:

```
dates <- seq(startdate, enddate, increment)
ts[dates]
```

The `window` function can select a range by start and end date:

```
window(ts, start = startdate, end = enddate)
```

Discussion

Recall our `xts` object that is a sample of inflation-adjusted IBM stock prices from the previous recipe:

```
head(ibm)
#>           ibm
#> 2000-01-01 78.6
#> 2000-01-03 82.0
#> 2000-01-04 79.2
#> 2000-01-05 82.0
#> 2000-01-06 80.6
#> 2000-01-07 80.2
```

We can select an observation by position, just like selecting elements from a vector (see [Recipe 2.9](#)):

```
ibm[2]
#>           ibm
#> 2000-01-03 82
```

We can also select multiple observations by position:

```
ibm[2:4]
#>           ibm
#> 2000-01-03 82.0
#> 2000-01-04 79.2
#> 2000-01-05 82.0
```

Sometimes it's more useful to select by date. Simply use the date itself as the index:

```
ibm[as.Date("2010-01-05")]
#>           ibm
#> 2010-01-05 103
```

Our `ibm` data is an `xts` object, so we can use date-like subsetting, too (the `zoo` object does not offer this flexibility):

```
ibm['2010-01-05']
ibm['20100105']
```

We can also select by a vector of Date objects:

```
dates <- seq(as.Date("2010-01-04"), as.Date("2010-01-08"), by = 2)
ibm[dates]
```

```
#>           ibm
#> 2010-01-04 104
#> 2010-01-06 102
#> 2010-01-08 103
```

The window function is easier for selecting a range of consecutive dates:

```
window(ibm, start = as.Date("2010-01-05"), end = as.Date("2010-01-07"))
#>           ibm
#> 2010-01-05 103
#> 2010-01-06 102
#> 2010-01-07 102
```

We can select a year/month combination using *yyyymm* subsetting:

```
ibm['201001'] # Jan 2010
```

Select year ranges using / like so:

```
ibm['2009/2011'] # all of 2009 - 2011
```

Or use / to select ranges including months:

```
ibm['2009/201001'] # all of 2009 plus Jan 2010
ibm['200906/201005'] # June 2009 through May 2010
```

See Also

The *xts* package provides many other clever ways to index a time series. See the package documentation.

14.5 Merging Several Time Series

Problem

You have two or more time series. You want to merge them into a single time series object.

Solution

Use a *zoo* or *xts* object to represent the time series, then use the *merge* function to combine them:

```
merge(ts1, ts2)
```

Discussion

Merging two time series is an incredible headache when the two series have differing timestamps. Consider these two time series, with the daily price of IBM stock from 1999 through 2017 and the monthly Consumer Price Index (CPI) for the same period:

```

load(file = "./data/ibm.rdata")
head(ibm)
#>          ibm
#> 1999-01-04 64.2
#> 1999-01-05 66.5
#> 1999-01-06 66.2
#> 1999-01-07 66.7
#> 1999-01-08 65.8
#> 1999-01-11 66.4
head(cpi)
#>          cpi
#> 1999-01-01 0.938
#> 1999-02-01 0.938
#> 1999-03-01 0.938
#> 1999-04-01 0.945
#> 1999-05-01 0.945
#> 1999-06-01 0.945

```

Obviously, the two time series have different timestamps because one is daily data and the other is monthly data. Even worse, the downloaded CPI data is timestamped for the first day of every month, even when that day is a holiday or weekend (e.g., New Year's Day).

Thank goodness for the `merge` function, which handles the messy details of reconciling the different dates:

```

head(merge(ibm, cpi))
#>          ibm   cpi
#> 1999-01-01   NA 0.938
#> 1999-01-04 64.2   NA
#> 1999-01-05 66.5   NA
#> 1999-01-06 66.2   NA
#> 1999-01-07 66.7   NA
#> 1999-01-08 65.8   NA

```

By default, `merge` finds the *union* of all dates: the output contains all dates from both inputs, and missing observations are filled with `NA` values. You can replace those `NA` values with the most recent observation by using the `na.locf` function from the `zoo` package:

```

head(na.locf(merge(ibm, cpi)))
#>          ibm   cpi
#> 1999-01-01   NA 0.938
#> 1999-01-04 64.2 0.938
#> 1999-01-05 66.5 0.938
#> 1999-01-06 66.2 0.938
#> 1999-01-07 66.7 0.938
#> 1999-01-08 65.8 0.938

```

(Here `locf` stands for “last observation carried forward.”) Observe that the NAs were replaced. However, `na.locf` left an NA in the first observation (1999-01-01) because there was no IBM stock price on that day.

You can get the *intersection* of all dates by setting `all = FALSE`:

```
head(merge(ibm, cpi, all = FALSE))
#>           ibm   cpi
#> 1999-02-01 63.1 0.938
#> 1999-03-01 59.2 0.938
#> 1999-04-01 62.3 0.945
#> 1999-06-01 79.0 0.945
#> 1999-07-01 92.4 0.949
#> 1999-09-01 89.8 0.956
```

Now the output is limited to observations that are *common* to both files.

Notice, however, that the intersection begins on February 1, not January 1. The reason is that January 1 is a holiday, so there is no IBM stock price for that date and hence no intersection with the CPI data. To fix this, see [Recipe 14.6](#).

14.6 Filling or Padding a Time Series

Problem

Your time series data is missing observations. You want to fill or pad the data with the missing dates/times.

Solution

Create a zero-width (dataless) `zoo` or `xts` object with the missing dates/times. Then merge your data with the zero-width object, taking the union of all dates:

```
empty <- zoo(, dates) # 'dates' is vector of the missing dates
merge(ts, empty, all = TRUE)
```

Discussion

The `zoo` package includes a handy feature in the constructor for `zoo` objects: you can omit the data and build a zero-width object. The object contains no data, just dates. We can use these “Frankenstein” objects to perform such operations as filling and padding on other time series objects.

Suppose you download monthly CPI data used in the last recipe. The data is timestamped with the first day of each month:

```
head(cpi)
#>           cpi
#> 1999-01-01 0.938
```

```
#> 1999-02-01 0.938
#> 1999-03-01 0.938
#> 1999-04-01 0.945
#> 1999-05-01 0.945
#> 1999-06-01 0.945
```

As far as R knows, we have no observations for the other days of the months. However, we know that each CPI value applies to the subsequent days through month-end. So first we build a zero-width object with every day of the decade, but no data:

```
dates <- seq(from = min(index(cpi)), to = max(index(cpi)), by = 1)
empty <- zoo(, dates)
```

We use `min(index(cpi))` and `max(index(cpi))` to get the minimum and maximum index values from our `cpi` data. So our resulting `empty` object is just an index of daily dates with the same range as our `cpi` data.

Then we take the union of the CPI data and the zero-width object, yielding a dataset filled with NA values:

```
filled.cpi <- merge(cpi, empty, all = TRUE)
head(filled.cpi)
#>           cpi
#> 1999-01-01 0.938
#> 1999-01-02    NA
#> 1999-01-03    NA
#> 1999-01-04    NA
#> 1999-01-05    NA
#> 1999-01-06    NA
```

The resulting time series contains every calendar day, with NAs where there was no observation. That might be what you need. However, a more common requirement is to replace each NA with the most recent observation as of that date. The `na.locf` function from the `zoo` package does exactly that:

```
filled.cpi <- na.locf(merge(cpi, empty, all = TRUE))
head(filled.cpi)
#>           cpi
#> 1999-01-01 0.938
#> 1999-01-02 0.938
#> 1999-01-03 0.938
#> 1999-01-04 0.938
#> 1999-01-05 0.938
#> 1999-01-06 0.938
```

January's value of 1 is carried forward until February 1, at which time it is replaced by the February value. Now every day has the latest CPI value as of that date. Note that in this dataset, the CPI is based on January 1, 1999 = 100% and all CPI values are relative to the value on that date:

```
tail(filled.cpi)
#>           cpi
```

```

#> 2017-11-26 1.41
#> 2017-11-27 1.41
#> 2017-11-28 1.41
#> 2017-11-29 1.41
#> 2017-11-30 1.41
#> 2017-12-01 1.41

```

We can use this recipe to fix the problem mentioned in [Recipe 14.5](#). There, the daily price of IBM stock and the monthly CPI data had no intersection on certain days. We can fix that using several different methods. One way is to pad the IBM data to include the CPI dates and then take the intersection (recall that `index(cpi)` returns all the dates in the CPI time series):

```

filled.ibm <- na.locf(merge(ibm, zoo(), index(cpi)))
head(merge(filled.ibm, cpi, all = FALSE))
#>           ibm   cpi
#> 1999-01-01  NA 0.938
#> 1999-02-01 63.1 0.938
#> 1999-03-01 59.2 0.938
#> 1999-04-01 62.3 0.945
#> 1999-05-01 73.6 0.945
#> 1999-06-01 79.0 0.945

```

That gives monthly observations. Another way is to fill out the CPI data (as described previously) and then take the intersection with the IBM data. That gives daily observations, as follows:

```

filled_data <- merge(ibm, filled.cpi, all = FALSE)
head(filled_data)
#>           ibm   cpi
#> 1999-01-04 64.2 0.938
#> 1999-01-05 66.5 0.938
#> 1999-01-06 66.2 0.938
#> 1999-01-07 66.7 0.938
#> 1999-01-08 65.8 0.938
#> 1999-01-11 66.4 0.938

```

Another common method for filling missing values uses the *cubic spline* technique, which interpolates smooth intermediate values from the known data. We can use the `zoo` function `na.spline` to fill our missing values using a cubic spline:

```

combined_data <- merge(ibm, cpi, all = TRUE)
head(combined_data)
#>           ibm   cpi
#> 1999-01-01  NA 0.938
#> 1999-01-04 64.2    NA
#> 1999-01-05 66.5    NA
#> 1999-01-06 66.2    NA
#> 1999-01-07 66.7    NA
#> 1999-01-08 65.8    NA

combined_spline <- na.spline(combined_data)

```

```
head(combined_spline)
#>           ibm   cpi
#> 1999-01-01  4.59 0.938
#> 1999-01-04 64.19 0.938
#> 1999-01-05 66.52 0.938
#> 1999-01-06 66.21 0.938
#> 1999-01-07 66.71 0.938
#> 1999-01-08 65.79 0.938
```

Notice that both the missing values for `cpi` and `ibm` were filled. However, the value filled in for January 1, 1999 for the `ibm` column seems out of line with the January 4th observation. This illustrates one of the challenges with cubic splines: they can become quite unstable if the value that is being interpolated is at the very beginning or the very end of a series. To get around this instability, we could get some data points from before January 1, 1999, then interpolate using `na.spline`, or we could simply choose a different interpolation method.

14.7 Lagging a Time Series

Problem

You want to shift a time series in time, either forward or backward.

Solution

Use the `lag` function. The second argument, `k`, is the number of periods to shift the data:

```
lag(ts, k)
```

Use positive `k` to shift the data forward in time (tomorrow's data becomes today's data). Use a negative `k` to shift the data backward in time (yesterday's data becomes today's data).

Discussion

Recall the `zoo` object containing five days of IBM stock prices from [Recipe 14.1](#):

```
ibm.daily
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>      132        131        130        130        131
```

To shift the data forward one day, we use `k = +1`:

```
lag(ibm.daily, k = +1, na.pad = TRUE)
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>      NA        132        131        130        130
```

We also set `na.pad = TRUE` to fill the trailing dates with NA. Otherwise, they would simply be dropped, resulting in a shortened time series.

To shift the data backward one day, we use `k = -1`. Again we use `na.pad = TRUE` to pad the beginning with NAs:

```
lag(ibm.daily, k = -1, na.pad = TRUE)
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>       NA         132         131         130         130
```

If the sign convention for `k` seems odd to you, you are not alone.



The function is called `lag`, but a positive `k` actually generates *leading* data, not lagging data. Use a negative `k` to get *lagging* data. Yes, this is bizarre. Perhaps the function should have been called `lead`.

The other thing to be careful with when using `lag` is that the `dplyr` package contains a function named `lag` as well. The arguments for `dplyr::lag` are not exactly the same as for the Base R `lag` function. In particular, `dplyr` uses `n` instead of `k`:

```
dplyr::lag(ibm.daily, n = 1)
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>       NA         132         131         130         130
```



If you want to load `dplyr`, you should use the namespace to be explicit about which `lag` function you are using. The Base R function is `stats::lag`, while the `dplyr` function is, naturally, `dplyr::lag`.

14.8 Computing Successive Differences

Problem

Given a time series, x , you want to compute the difference between successive observations: $(x_2 - x_1)$, $(x_3 - x_2)$, $(x_4 - x_3)$,

Solution

Use the `diff` function:

```
diff(x)
```

Discussion

The `diff` function is generic, so it works on simple vectors, `xts` objects, and `zoo` objects. The beauty of differencing a `zoo` or `xts` object is that the result is the same type of object you started with *and* the differences have the correct dates. Here we compute the differences for successive prices of IBM stock:

```
ibm.daily
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>      132       131       130       130       131
diff(ibm.daily)
#> 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>     -1.60     -0.85     -0.45      1.30
```

The difference labeled 2010-01-05 is the change from the previous day (2010-01-04), which is usually what you want. The differenced series is shorter than the original series by one element because R can't compute the change as of 2010-01-04, of course.

By default, `diff` computes successive differences. You can compute differences that are more widely spaced by using its `lag` parameter. Suppose you have monthly CPI data and want to compute the change from the previous 12 months, giving the year-over-year change. Specify a `lag` of 12:

```
head(cpi, 24)
#>                  cpi
#> 1999-01-01 0.938
#> 1999-02-01 0.938
#> 1999-03-01 0.938
#> 1999-04-01 0.945
#> 1999-05-01 0.945
#> 1999-06-01 0.945
#> 1999-07-01 0.949
#> 1999-08-01 0.952
#> 1999-09-01 0.956
#> 1999-10-01 0.957
#> 1999-11-01 0.959
#> 1999-12-01 0.961
#> 2000-01-01 0.964
#> 2000-02-01 0.968
#> 2000-03-01 0.974
#> 2000-04-01 0.973
#> 2000-05-01 0.975
#> 2000-06-01 0.981
#> 2000-07-01 0.983
#> 2000-08-01 0.983
#> 2000-09-01 0.989
#> 2000-10-01 0.990
#> 2000-11-01 0.992
#> 2000-12-01 0.994
head(diff(cpi, lag = 12), 24) # Compute year-over-year change
#>                  cpi
```

```
#> 1999-01-01      NA
#> 1999-02-01      NA
#> 1999-03-01      NA
#> 1999-04-01      NA
#> 1999-05-01      NA
#> 1999-06-01      NA
#> 1999-07-01      NA
#> 1999-08-01      NA
#> 1999-09-01      NA
#> 1999-10-01      NA
#> 1999-11-01      NA
#> 1999-12-01      NA
#> 2000-01-01 0.0262
#> 2000-02-01 0.0302
#> 2000-03-01 0.0353
#> 2000-04-01 0.0285
#> 2000-05-01 0.0296
#> 2000-06-01 0.0353
#> 2000-07-01 0.0342
#> 2000-08-01 0.0319
#> 2000-09-01 0.0330
#> 2000-10-01 0.0330
#> 2000-11-01 0.0330
#> 2000-12-01 0.0330
```

14.9 Performing Calculations on Time Series

Problem

You want to use arithmetic and common functions on time series data.

Solution

No problem. R is pretty clever about operations on `zoo` and `xts` objects. You can use arithmetic operators (`+`, `-`, `*`, `/`, etc.) as well as common functions (`sqrt`, `log`, etc.) and usually get what you expect.

Discussion

When you perform arithmetic on `zoo` or `xts` objects, R aligns the objects according to date so that the results make sense. Suppose we want to compute the percentage change in IBM stock. We need to divide the daily change by the price, but those two time series are not naturally aligned—they have different start times and different lengths. Here's an illustration with a `zoo` object:

```
ibm.daily
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08
#>       132       131       130       130       131
diff(ibm.daily)
```

```
#> 2010-01-05 2010-01-06 2010-01-07 2010-01-08  
#> -1.60      -0.85      -0.45      1.30
```

Fortunately, when we divide one series by the other, R aligns the series for us and returns a `zoo` object:

```
diff(ibm.daily) / ibm.daily  
#> 2010-01-05 2010-01-06 2010-01-07 2010-01-08  
#> -0.01223   -0.00654   -0.00347   0.00994
```

We can scale the result by 100 to compute the percentage change, and the result is another `zoo` object:

```
100 * (diff(ibm.daily) / ibm.daily)  
#> 2010-01-05 2010-01-06 2010-01-07 2010-01-08  
#> -1.223     -0.654     -0.347     0.994
```

Functions work just as well. If we compute the logarithm or square root of a `zoo` object, the result is a `zoo` object with the timestamps preserved:

```
log(ibm.daily)  
#> 2010-01-04 2010-01-05 2010-01-06 2010-01-07 2010-01-08  
#> 4.89       4.87       4.87       4.86       4.87
```

In investment management, computing the difference of logarithms of prices is quite common. That's a piece of cake in R:

```
diff(log(ibm.daily))  
#> 2010-01-05 2010-01-06 2010-01-07 2010-01-08  
#> -0.01215   -0.00652   -0.00347   0.00998
```

See Also

See [Recipe 14.8](#) for the special case of computing the difference between successive values.

14.10 Computing a Moving Average

Problem

You want to compute the moving average of a time series.

Solution

Use the `rollmean` function of the `zoo` package to calculate the k -period moving average:

```
library(zoo)  
ma <- rollmean(ts, k)
```

Here `ts` is the time series data, captured in a `zoo` object, and `k` is the number of periods.

For most financial applications, you want `rollmean` to calculate the mean using only historical data; that is, for each day, you use only the data available that day. To do that, specify `align = "right"`. Otherwise, `rollmean` will “cheat” and use future data that was actually unavailable at the time:

```
ma <- rollmean(ts, k, align = "right")
```

Discussion

Traders are fond of moving averages for smoothing out fluctuations in prices. The formal name is the *rolling mean*. You could calculate the rolling mean as described in [Recipe 14.12](#) by combining the `rollapply` function and the `mean` function, but `rollmean` is much faster.

Besides speed, the beauty of `rollmean` is that it returns the same type of time series object it's called on (i.e., `xts` or `zoo`). For each element in the object, its date is the “as of” date for a calculated mean. Because the result is a time series object, you can easily merge the original data and the moving average and then plot them together as in [Figure 14-3](#):

```
ibm_year <- ibm["2016"]
ma_ibm <- rollmean(ibm_year, 7, align = "right")
ma_ibm <- merge(ma_ibm, ibm_year)
plot(ma_ibm)
```

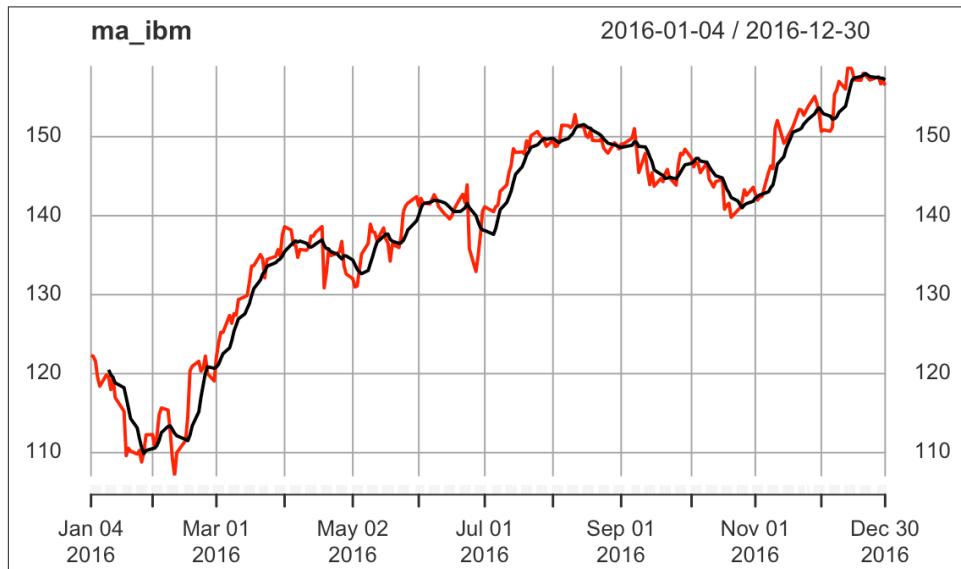


Figure 14-3. Rolling average plot

The output is normally missing a few initial data points, since `rollmean` needs a full k observations to compute the mean. Consequently, the output is shorter than the input. If that's a problem, specify `na.pad = TRUE`; then `rollmean` will pad the initial output with NA values.

See Also

See [Recipe 14.12](#) for more about the `align` parameter.

The moving average described here is a simple moving average. The `quantmod`, `TTR`, and `fTrading` packages contain functions for computing and plotting many kinds of moving averages, including simple ones.

14.11 Applying a Function by Calendar Period

Problem

Given a time series, you want to group the contents by a calendar period (e.g., week, month, or year) and then apply a function to each group.

Solution

The `xts` package includes functions for processing a time series by day, week, month, quarter, or year:

```
apply.daily(ts, f)
apply.weekly(ts, f)
apply.monthly(ts, f)
apply.quarterly(ts, f)
apply.yearly(ts, f)
```

Here `ts` is an `xts` time series, and `f` is the function to apply to each day, week, month, quarter, or year.

If your time series is a `zoo` object, convert it to an `xts` object first so you can access these functions; for example:

```
apply.monthly(as.xts(ts), f)
```

Discussion

It is common to process time series data according to calendar period. But figuring calendar periods is tedious at best and bizarre at worst. Let these functions do the heavy lifting.

Suppose we have a five-year history of IBM stock prices stored in an `xts` object:

```

ibm_5 <- ibm["2012/2017"]
head(ibm_5)
#>           ibm
#> 2012-01-03 152
#> 2012-01-04 151
#> 2012-01-05 150
#> 2012-01-06 149
#> 2012-01-09 148
#> 2012-01-10 148

```

We can calculate the average price by month if we use `apply.monthly` and `mean` together:

```

ibm_mm <- apply.monthly(ibm_5, mean)
head(ibm_mm)
#>           ibm
#> 2012-01-31 151
#> 2012-02-29 158
#> 2012-03-30 166
#> 2012-04-30 167
#> 2012-05-31 164
#> 2012-06-29 159

```

Notice that the IBM data is in an `xts` object from the start. Had the data been in a `zoo` object, we would have needed to convert it to `xts` using `as.xts`.

A more interesting application is calculating volatility by calendar month, where volatility is measured as the standard deviation of daily log-returns. Daily log-returns are calculated this way:

```
diff(log(ibm_5))
```

We calculate their standard deviation, month by month, like this:

```
apply.monthly(as.xts(diff(log(ibm_5))), sd)
```

We can scale the daily number to estimate annualized volatility, as shown in Figure 14-4:

```

ibm_vol <- sqrt(251) * apply.monthly(as.xts(diff(log(ibm_5))), sd)
plot(ibm_vol,
     main = "IBM: Monthly Volatility"
)

```

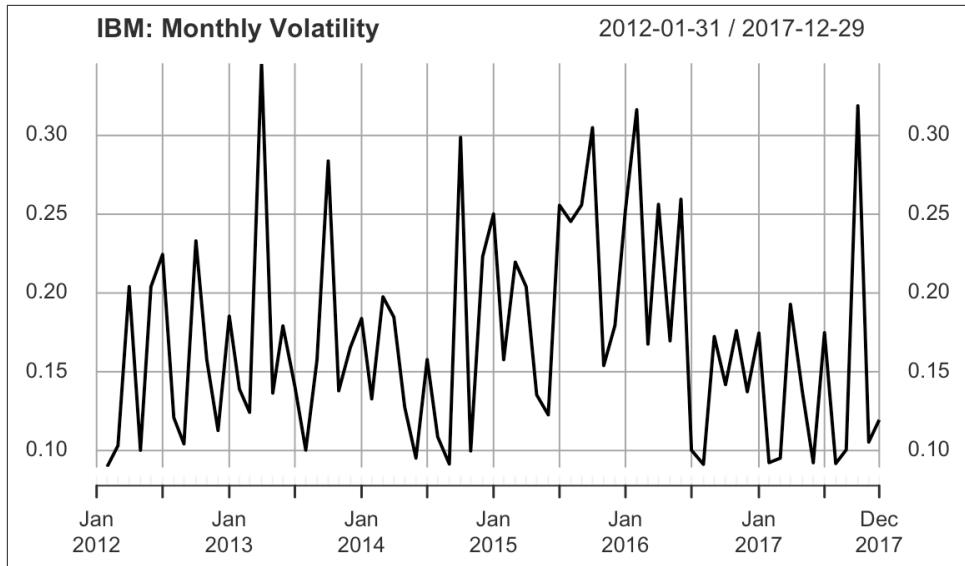


Figure 14-4. IBM volatility plot

14.12 Applying a Rolling Function

Problem

You want to apply a function to a time series in a rolling manner: calculate the function at a data point using some window of time around that point, move to the next data point, calculate the function around that point, move to the next data point, and so forth.

Solution

Use the `rollapply` function in the `zoo` package. The `width` parameter defines how many data points from the time series (`ts`) should be processed by the function (`f`) at each point:

```
library(zoo)
rollapply(ts, width, f)
```

For many applications, you will likely set `align = "right"` to avoid computing `f` with historical data that was unavailable at the time:

```
rollapply(ts, width, f, align = "right")
```

Discussion

The `rollapply` function extracts a “window” of data from your time series, calls your function with that data, saves the result, and moves to the next window—and repeats this pattern for the entire input. As an illustration, consider calling `rollapply` with a width of 21:

```
rollapply(ts, 21, f)
```

`rollapply` will repeatedly call the function, `f`, with a sliding window of data, like this:

1. `f(ts[1:21])`
2. `f(ts[2:22])`
3. `f(ts[3:23])`
4. ... etc. ...

Observe that the function should expect one argument, which is a vector of values. `rollapply` will save the returned values before packaging them into a `zoo` object along with a timestamp for every value. The choice of timestamp depends on the `align` parameter given to `rollapply`:

`align="right"`

The timestamp is taken from the rightmost value.

`align="left"`

The timestamp is taken from the leftmost value.

`align="center" (default)`

The timestamp is taken from the middle value.

By default, `rollapply` will recalculate the function at successive data points. You may instead want to calculate the function at every n th data point. Use the `by = n` parameter to have `rollapply` move ahead n points after each function call. When we calculate the rolling standard deviation of a time series, for example, we usually want each window of data to be separate, not overlapping, so we set the `by` value equal to the window size:

```
ibm_sds <- rollapply(ibm_5, width = 30, FUN = sd, by = 30, align = "right")
ibm_sds <- na.omit(ibm_sds)
head(ibm_sds)
```

The `rollapply` function will, by default, return an object with as many observations as your input data with the missing values filled with `NA`. In the preceding example we use `na.omit` to drop the `NA` values so that our resulting object has records only for the dates for which we have values.

14.13 Plotting the Autocorrelation Function

Problem

You want to plot the autocorrelation function (ACF) of your time series.

Solution

Use the `acf` function:

```
acf(ts)
```

Discussion

The autocorrelation function is an important tool for revealing the interrelationships within a time series. It is a collection of correlations, ρ_k for $k = 1, 2, 3, \dots$, where ρ_k is the correlation between all pairs of data points that are exactly k steps apart.

Visualizing the autocorrelations is much more useful than listing them, so the `acf` function plots them for each value of k . The following example shows the autocorrelation functions for two time series, one with autocorrelations (Figure 14-5) and one without (Figure 14-6). The dashed line delimits the significant and insignificant correlations: values above the line are significant (the height of the line is determined by the amount of data). We can plot them as follows:

```
load(file = "./data/ts_acf.rdata")
acf(ts1, main = "Significant Autocorrelations")
acf(ts2, main = "Insignificant Autocorrelations")
```

Significant Autocorrelations

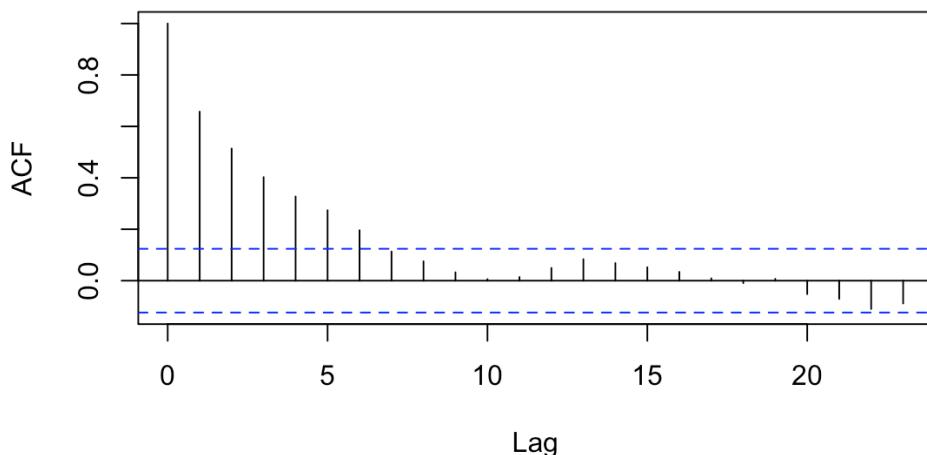


Figure 14-5. Autocorrelations at each lag: $ts1$

Insignificant Autocorrelations

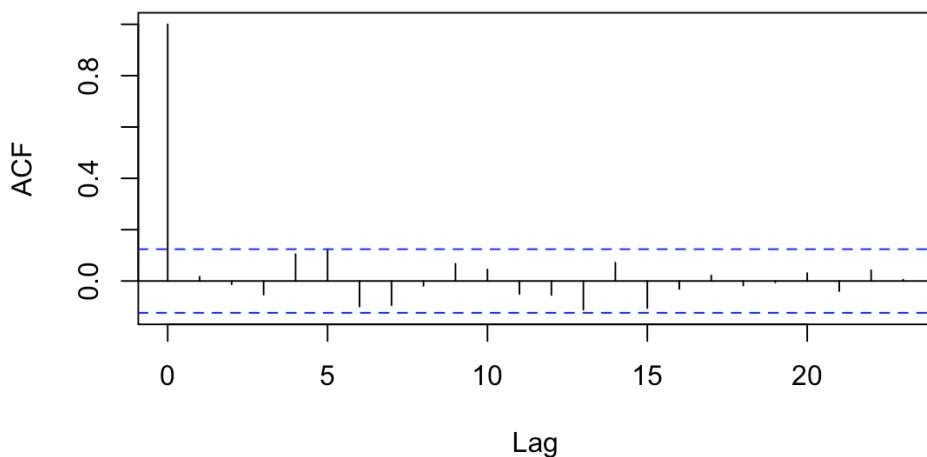


Figure 14-6. Autocorrelations at each lag: $ts2$

The presence of autocorrelations is one indication that an autoregressive integrated moving average (ARIMA) model could model the time series. From the ACF, you can count the number of significant autocorrelations, which is a useful estimate of the number of moving average (MA) coefficients in the model. Figure 14-5 shows seven

significant autocorrelations, for example, so we estimate that its ARIMA model will require seven MA coefficients (MA(7)). That estimate is just a starting point, however, and must be verified by fitting and diagnosing the model.

14.14 Testing a Time Series for Autocorrelation

Problem

You want to test your time series for the presence of autocorrelations.

Solution

Use the `Box.test` function, which implements the Box–Pierce test for autocorrelation:

```
Box.test(ts)
```

The output includes a p -value. Conventionally, a p -value of less than 0.05 indicates that the data contains significant autocorrelations, whereas a p -value exceeding 0.05 provides no such evidence.

Discussion

Graphing the autocorrelation function is useful for digging into your data. Sometimes, however, you just need to know whether or not the data is autocorrelated. A statistical test such as the Box–Pierce test can provide an answer.

We can apply the Box–Pierce test to the data whose autocorrelation function we plotted in [Recipe 14.13](#). The test shows p -values for the two time series that are nearly 0 and 0.79, respectively:

```
Box.test(ts1)
#>
#> Box-Pierce test
#>
#> data: ts1
#> X-squared = 100, df = 1, p-value <2e-16

Box.test(ts2)
#>
#> Box-Pierce test
#>
#> data: ts2
#> X-squared = 0.07, df = 1, p-value = 0.8
```

The p -value near 0 indicates that the first time series has significant autocorrelations. (We don't know which autocorrelations are significant; we just know they exist.) The

p-value of 0.8 indicates that the test did not detect autocorrelations in the second time series.

The `Box.test` function can also perform the Ljung–Box test, which is better for small samples. That test calculates a *p*-value whose interpretation is the same as that for the Box–Pierce *p*-value:

```
Box.test(ts, type = "Ljung-Box")
```

See Also

See [Recipe 14.13](#) to plot the autocorrelation function, a visual check of the autocorrelation.

14.15 Plotting the Partial Autocorrelation Function

Problem

You want to plot the partial autocorrelation function (PACF) for your time series.

Solution

Use the `pacf` function:

```
pacf(ts)
```

Discussion

The partial autocorrelation function is another tool for revealing the interrelationships in a time series. However, its interpretation is much less intuitive than that of the autocorrelation function. We'll leave the mathematical definition of partial correlation to a textbook on statistics. Here, we'll just say that the partial correlation between two random variables, X and Y , is the correlation that remains after accounting for the correlation shown by X and Y with all other variables. In the case of time series, the partial autocorrelation at lag k is the correlation between all data points that are exactly k steps apart, after accounting for their correlation with the data between those k steps.

The practical value of a PACF is that it helps you to identify the number of autoregression (AR) coefficients in an ARIMA model. The following example shows the PACF for the two time series used in [Recipe 14.13](#). One of these series has partial autocorrelations and one does not. Lag values whose lines cross above the dotted line are statistically significant. In the first time series ([Figure 14-7](#)) there are two such values, at $k = 1$ and $k = 2$, so our initial ARIMA model will have two AR coefficients (AR(2)). As with autocorrelation, however, that is just an initial estimate and must be

verified by fitting and diagnosing the model. The second time series (Figure 14-8) shows no such autocorrelation pattern. We can plot them as follows:

```
pacf(ts1, main = "Significant Partial Autocorrelations")  
pacf(ts2, main = "Insignificant Partial Autocorrelations")
```

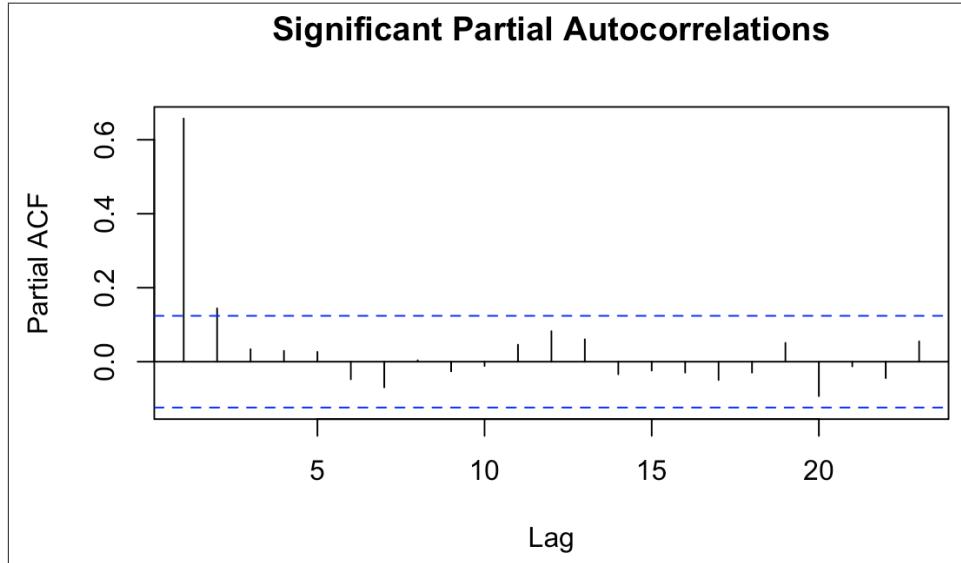


Figure 14-7. Autocorrelations at each lag: ts1

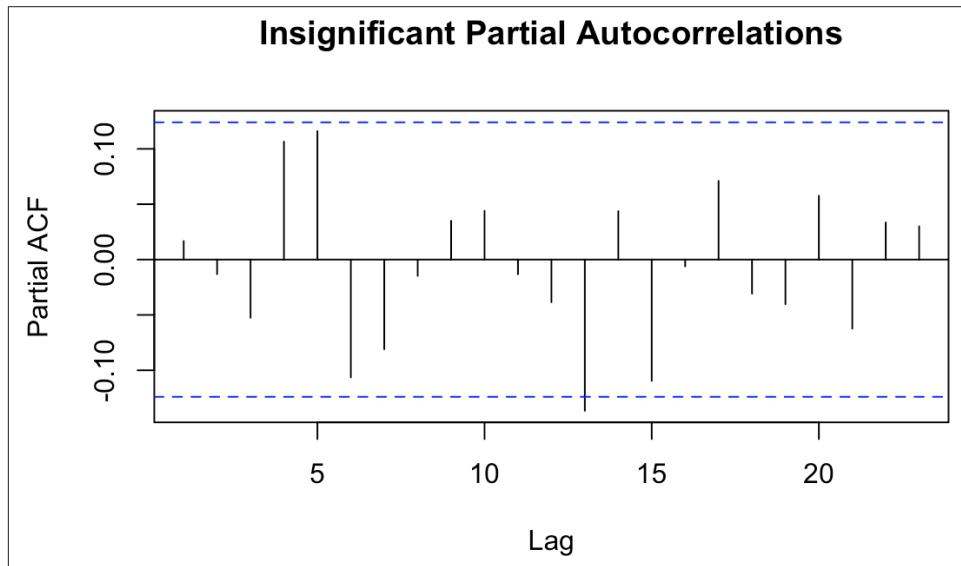


Figure 14-8. Autocorrelations at each lag: ts2

See Also

See Recipe 14.13.

14.16 Finding Lagged Correlations Between Two Time Series

Problem

You have two time series, and you are wondering if there is a lagged correlation between them.

Solution

Use the `Ccf` function from the package `forecast` to plot the cross-correlation function, which will reveal lagged correlations:

```
library(forecast)
Ccf(ts1, ts2)
```

Discussion

The cross-correlation function helps you discover lagged correlations between two time series. A lagged correlation occurs when today's value in one time series is correlated with a future or past value in the other time series.

Consider the relationship between commodity prices and bond prices. Some analysts believe those prices are connected because changes in commodity prices are a barometer of inflation, one of the key factors in bond pricing. Can we discover a correlation between them?

Figure 14-9 shows a cross-correlation function generated from daily changes in bond prices and a commodity price index:¹

```
library(forecast)
load(file = "./data/bnd_cmtv.Rdata")
b <- coredata(bonds)[, 1]
c <- coredata(cmdtys)[, 1]

Ccf(b, c, main = "Bonds vs. Commodities")
```

¹ Specifically, the `bonds` variable is the log-returns of the Vanguard Long-Term Bond Index Fund (VBLTX), and the `cmdtys` variable is the log-returns of the Invesco DB Commodity Tracking Fund (DBC). The data was taken from the period 2007-01-01 through 2017-12-31.

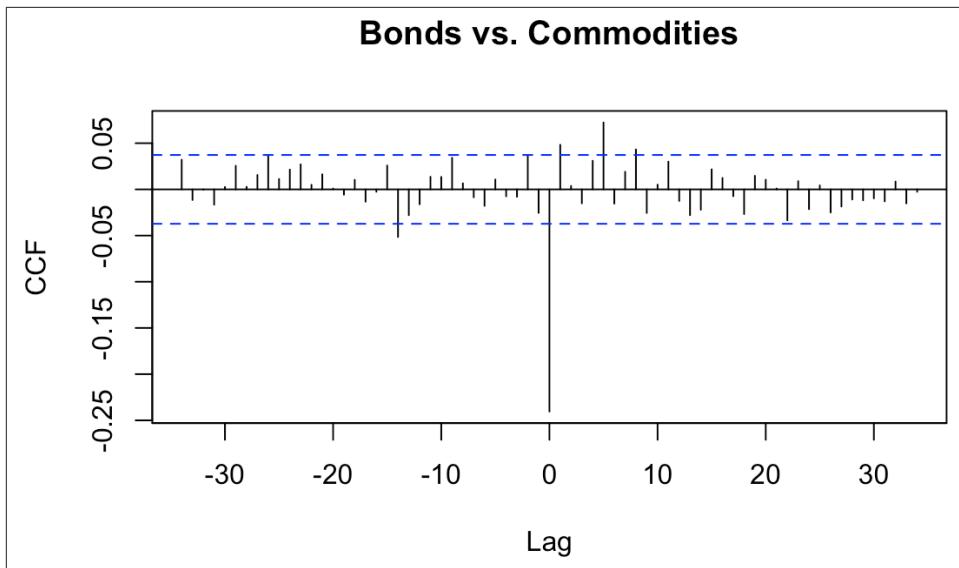


Figure 14-9. Cross-correlation function

Note that since the objects we start with, `bonds` and `cmdtys`, are `xts` objects, we extract from each the vector of data using `coredata()[1]`. This is because the `Ccf` function expects inputs to be simple vectors.

Every vertical line shows the correlation between the two time series at some lag, as indicated along the x-axis. If a correlation extends above or below the dotted lines, it is statistically significant.

Notice that the correlation at lag 0 is -0.24 , which is the simple correlation between the variables:

```
cor(b, c)
#> [1] -0.24
```

Much more interesting are the correlations at lags 1, 5, and 8, which are statistically significant. Evidently there is some “ripple effect” in the day-to-day prices of bonds and commodities because changes today are correlated with changes tomorrow. Discovering this sort of relationship is useful to short-term forecasters such as market analysts and bond traders.

14.17 Detrending a Time Series

Problem

Your time series data contains a trend that you want to remove.

Solution

Use linear regression to identify the trend component, and then subtract the trend component from the original time series. These two lines show how to detrend the `zoo` object `ts` and put the result in `detr`:

```
m <- lm(coredata(ts) ~ index(ts))
detr <- zoo(resid(m), index(ts))
```

Discussion

Some time series data contains trends, which means that it gradually slopes upward or downward over time. Suppose our time series object (a `zoo` object in this case), `yield`, contains a trend as shown in Figure 14-10.

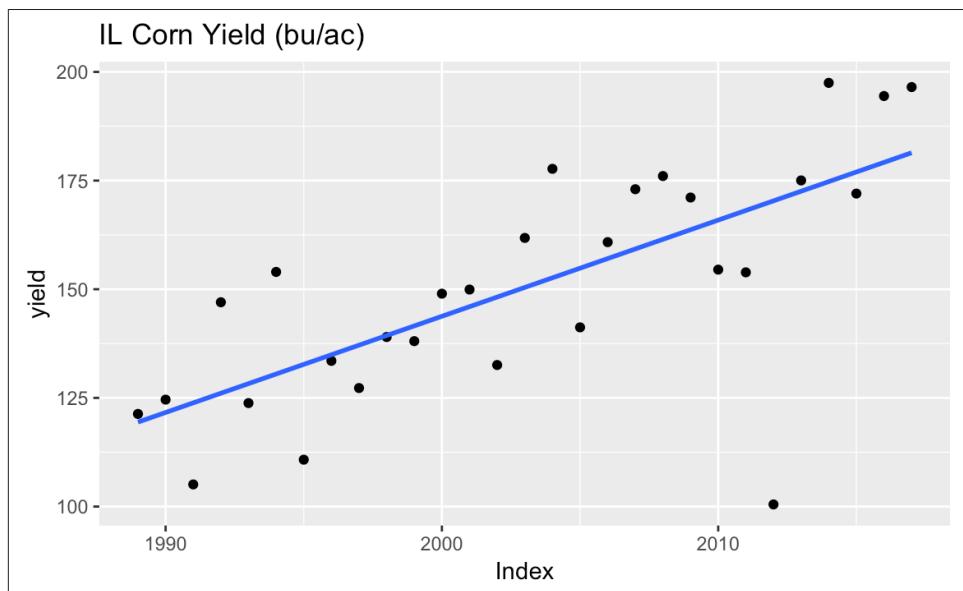


Figure 14-10. Time series with trend

We can remove the trend component in two steps. First, we identify the overall trend by using the linear model function, `lm`. The model should use the time series index for the `x` variable and the time series data for the `y` variable:

```
m <- lm(coredata(yield) ~ index(yield))
```

Second, we remove the linear trend from the original data by subtracting the straight line found by `lm`. This is easy because we have access to the linear model's residuals, which are defined by the difference between the original data and the fitted line:

$$r_i = y_i - \beta_1 x_i - \beta_0$$

where r_i is the i th residual and β_1 and β_0 are the model's slope and intercept, respectively. We can extract the residuals from the linear model by using the `resid` function and then embed the residuals inside a `zoo` object:

```
detr <- zoo(resid(m), index(yield))
```

Notice that we use the same time index as the original data. When we plot `detr` it is clearly trendless, as is evident in [Figure 14-11](#):

```
autoplot(detr)
```

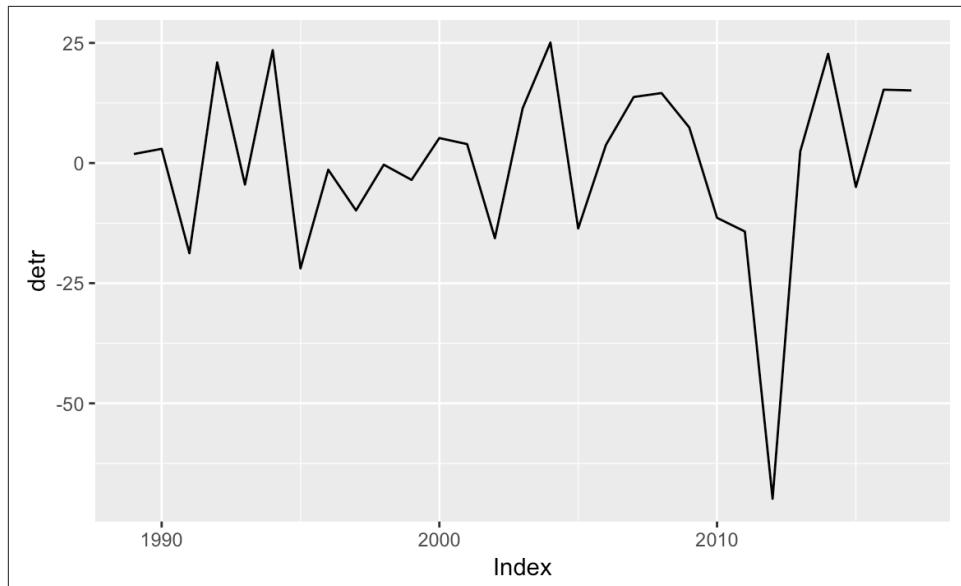


Figure 14-11. Residual plot

This data is the state average corn yield for Illinois in bushels per acre (bu/ac), so `detr` is the difference between the actual yield and the trend. Sometimes when detrending you may want to determine the percent deviation from the trend. In that case you can divide by the initial measure (see [Figure 14-12](#)):

```
library(patchwork)
# y <- autoplot(yield) +
#   labs(x='Year', y='Yield (bu/ac)', title='IL Corn Yield')
d <- autoplot(detr, geom = "point") +
  labs(
    x = "Year", y = "Yield Dev (bu/ac)",
    title = "IL Corn Yield Deviation from Trend (bu/ac)"
  )
dp <- autoplot(detr / yield, geom = "point") +
```

```

  labs(
    x = "Year", y = "Yield Dev (%)",
    title = "IL Corn Yield Deviation from Trend (%)"
  )

```

d / dp

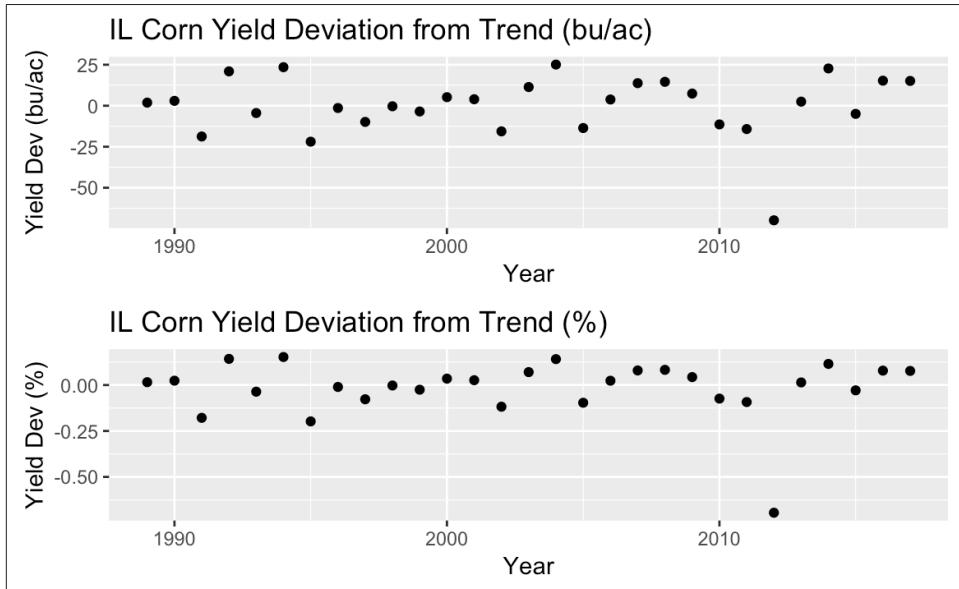


Figure 14-12. Detrended plots

The top plot in Figure 14-12 shows the yield deviation from the trend in bu/ac (the original units), while the lower plot shows the percent deviation from the trend.

14.18 Fitting an ARIMA Model

Problem

You want to fit an ARIMA model to your time series data.

Solution

The `auto.arima` function in the `forecast` package can select the correct model order and fit the model to your data:

```

library(forecast)
auto.arima(x)

```

If you already know the model order, (p, d, q) , then the `arima` function can fit the model directly:

```
arima(x, order = c(p, d, q))
```

Discussion

Creating an ARIMA model involves three steps:

1. Identify the model order.
2. Fit the model to the data, giving the coefficients.
3. Apply diagnostic measures to validate the model.

The model order is usually denoted by three integers, (p, d, q) , where p is the number of autoregressive coefficients, d is the degree of differencing, and q is the number of moving average coefficients.

When most of us build an ARIMA model, we are usually clueless about the appropriate order. Rather than tediously searching for the best combination of p , d , and q , we typically use `auto.arima`, which does the searching for us:

```
library(forecast)
library(fpp2) # for example data

auto.arima(ausbeer)
#> Series: ausbeer
#> ARIMA(1,1,2)(0,1,1)[4]
#>
#> Coefficients:
#>       ar1     ma1     ma2    sma1
#>      0.050   -1.009   0.375  -0.743
#> s.e.  0.196   0.183   0.153   0.050
#>
#> sigma^2 estimated as 241: log likelihood=-886
#> AIC=1783  AICc=1783  BIC=1800
```

In this case, `auto.arima` decided the best order was $(1, 1, 2)$, which means that it differenced the data once ($d = 1$) before selecting a model with one AR coefficient ($p = 1$) and two MA coefficients ($q = 2$). In addition, the `auto.arima` function determined that our data has seasonality and included the seasonal terms $P = 0$, $D = 1$, $Q = 1$ and a period of $m = 4$. The seasonality terms are similar to the nonseasonal ARIMA terms, but relate to the seasonality component of the model. The m term tells us the periodicity of the seasonality, which in this case is quarterly. We can see this more easily if we plot the `ausbeer` data as in [Figure 14-13](#):

```
autoplot(ausbeer)
```

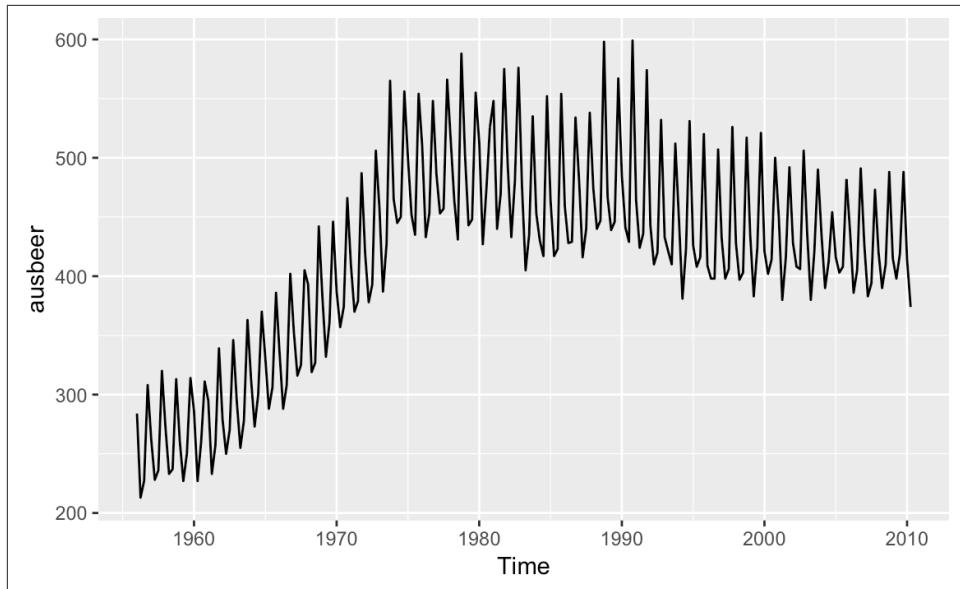


Figure 14-13. Australian beer consumption

By default, `auto.arima` limits p and q to the range $0 \leq p \leq 5$ and $0 \leq q \leq 5$. If you are confident that your model needs fewer than five coefficients, use the `max.p` and `max.q` parameters to limit the search further; this makes it faster. Likewise, if you believe that your model needs more coefficients, use `max.p` and `max.q` to expand the search limits.

If you want to turn off the seasonality component of `auto.arima`, you can set `seasonal = FALSE`:

```
auto.arima(ausbeer, seasonal = FALSE)
#> Series: ausbeer
#> ARIMA(3,2,2)
#>
#> Coefficients:
#>       ar1      ar2      ar3      ma1      ma2
#>     -0.957   -0.987   -0.925  -1.043   0.142
#> s.e.   0.026    0.018    0.024    0.062   0.062
#>
#> sigma^2 estimated as 327:  log likelihood=-935
#> AIC=1882  AICc=1882  BIC=1902
```

But notice that since the model fits a nonseasonal model, the coefficients are different than in the seasonal model.

If you already know the order of your ARIMA model, the `arima` function can quickly fit the model to your data:

```

arima(ausbeer, order = c(3, 2, 2))
#>
#> Call:
#> arima(x = ausbeer, order = c(3, 2, 2))
#>
#> Coefficients:
#>      ar1      ar2      ar3      ma1      ma2
#>     -0.957   -0.987   -0.925   -1.043   0.142
#> s.e.   0.026    0.018    0.024    0.062   0.062
#>
#> sigma^2 estimated as 319:  log likelihood = -935,  aic = 1882

```

The output looks identical to that of `auto.arima` with the `seasonal` parameter set to `FALSE`. What you can't see here is that `arima` executes much more quickly.

The output from `auto.arima` and `arima` includes the fitted coefficients and the standard error (`s.e.`) for each coefficient:

Coefficients:					
	ar1	ar2	ar3	ma1	ma2
#>	-0.9569	-0.9872	-0.9247	-1.0425	0.1416
s.e.	0.0257	0.0184	0.0242	0.0619	0.0623

You can find the coefficients' confidence intervals by capturing the ARIMA model in an object and then using the `confint` function:

```

m <- arima(x = ausbeer, order = c(3, 2, 2))
confint(m)
#>      2.5 % 97.5 %
#> ar1 -1.0072 -0.907
#> ar2 -1.0232 -0.951
#> ar3 -0.9721 -0.877
#> ma1 -1.1639 -0.921
#> ma2  0.0195  0.264

```

This output illustrates a major headache of ARIMA modeling: not all the coefficients are necessarily significant. If one of the intervals contains zero, the true coefficient might be zero itself, in which case the term is unnecessary.

If you discover that your model contains insignificant coefficients, use [Recipe 14.19](#) to remove them.



The `auto.arima` and `arima` functions contain useful features for fitting the best model. For example, you can force them to include or exclude a trend component. See the help pages for details.

A final caveat: the danger of `auto.arima` is that it makes ARIMA modeling look simple. ARIMA modeling is *not* simple. It is more art than science, and the automatically

generated model is just a starting point. We urge you to review a good book about ARIMA modeling before settling on a final model.

See Also

See [Recipe 14.20](#) for performing diagnostic tests on the ARIMA model.

As a textbook on time series forecasting we highly recommend *Forecasting: Principles and Practice*, 2nd ed., by Rob J. Hyndman and George Athanasopoulos, which is [freely available online](#).

14.19 Removing Insignificant ARIMA Coefficients

Problem

One or more of the coefficients in your ARIMA model are statistically insignificant. You want to remove them.

Solution

The `arima` function includes the parameter `fixed`, which is a vector. The vector should contain one element for every coefficient in the model, including a term for the drift (if any). Each element is either `NA` or `0`. Use `NA` for the coefficients to be kept and use `0` for the coefficients to be removed. This example shows an ARIMA(2, 1, 2) model with the first AR coefficient and the first MA coefficient forced to be `0`:

```
arima(x, order = c(2, 1, 2), fixed = c(0, NA, 0, NA))
```

Discussion

The `fpp2` package contains a dataset called `euretail`, which is a quarterly retail index for the Euro area. Let's run `auto.arima` on the data and look at the 98% confidence intervals:

```
m <- auto.arima(euretail)
#
#> Series: euretail
#> ARIMA(0,1,3)(0,1,1)[4]
#
#> Coefficients:
#>          ma1     ma2     ma3     sma1
#>        0.263   0.369   0.420   -0.664
#> s.e.  0.124   0.126   0.129   0.155
#
#> sigma^2 estimated as 0.156:  log likelihood=-28.6
#> AIC=67.3  AICc=68.4  BIC=77.7
confint(m, level = .98)
#>      1 %    99 %
```

```
#> ma1 -0.0246 0.551
#> ma2  0.0774 0.661
#> ma3  0.1190 0.721
#> sma1 -1.0231 -0.304
```

In this example, we can see that the 98% confidence interval for the `ma1` parameter contains 0 and we can reasonably conclude that this parameter is insignificant at this level of confidence. We can set this parameter to 0 using the `fixed` parameter:

```
m <- arima(euretail,
            order = c(0, 1, 3),
            seasonal = c(0, 1, 1),
            fixed = c(0, NA, NA, NA)).
```

`m`
#>
#> Call:
#> arima(x = euretail,
#> order = c(0, 1, 3),
#> seasonal = c(0, 1, 1),
#> fixed = c(0,
#> NA, NA, NA))
#>
#> Coefficients:
#> ma1 ma2 ma3 sma1
#> 0 0.404 0.293 -0.700
#> s.e. 0 0.129 0.107 0.135
#>
#> sigma^2 estimated as 0.156: log likelihood = -30.8, aic = 69.5

Observe that the `ma1` coefficient is now 0. The remaining coefficients (`ma2`, `ma3`, `sma1`) are still significant, as shown by their confidence intervals, so we have a reasonable model:

```
confint(m, level = .98)
#>      1 %    99 %
#> ma1      NA      NA
#> ma2  0.1049  0.703
#> ma3  0.0438  0.542
#> sma1 -1.0140 -0.386
```

14.20 Running Diagnostics on an ARIMA Model

Problem

You have built an ARIMA model using the `forecast` package, and you want to run diagnostic tests to validate the model.

Solution

Use the `checkresiduals` function. This example fits the ARIMA model using `auto.arima`, puts the results in `m`, and then runs diagnostics on the model:

```
m <- auto.arima(x)
checkresiduals(m)
```

Discussion

The result of `checkresiduals` is a set of three graphs, as shown in [Figure 14-14](#). A good model should produce results like these:

```
#>
#> Ljung-Box test
#>
#> data: Residuals from ARIMA(1,1,2)(0,1,1)[4]
#> Q* = 5, df = 4, p-value = 0.3
#>
#> Model df: 4. Total lags used: 8
```

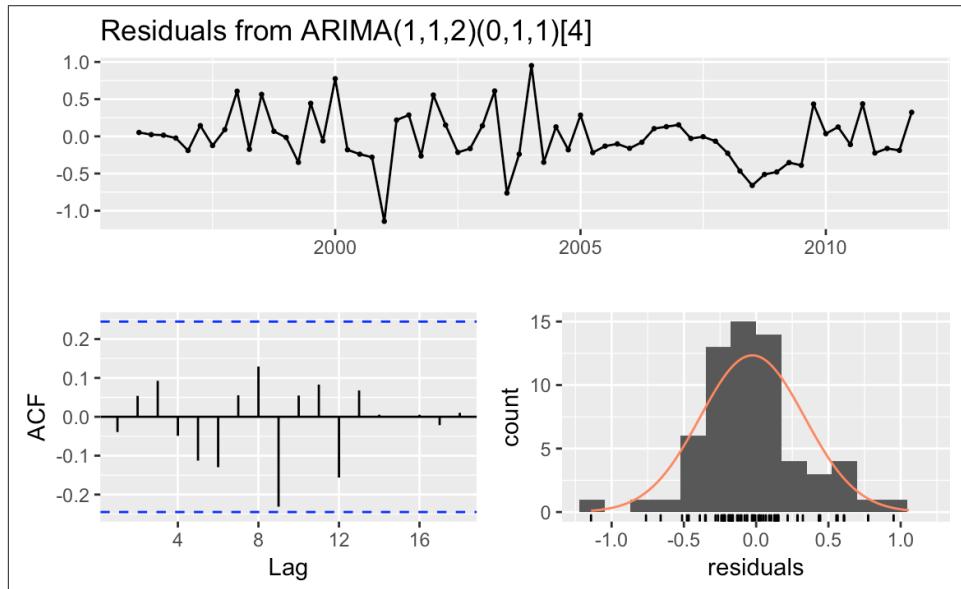


Figure 14-14. Residuals plots: good model

Here's what's good about the graphs:

- The standardized residuals don't show clusters of volatility.
- The autocorrelation function (ACF) shows no significant autocorrelation between the residuals.

- The residuals look bell-shaped, suggesting they are reasonably symmetrical.
- The p -value in the Ljung–Box test is large, indicating that the residuals are patternless—meaning all the information has been extracted by the model and only noise is left behind.

For contrast, Figure 14-15 shows diagnostic charts with problems:

```
#>
#> Ljung-Box test
#>
#> data: Residuals from ARIMA(1,1,1)(0,0,1)[4]
#> Q* = 20, df = 5, p-value = 5e-04
#>
#> Model df: 3. Total lags used: 8
```

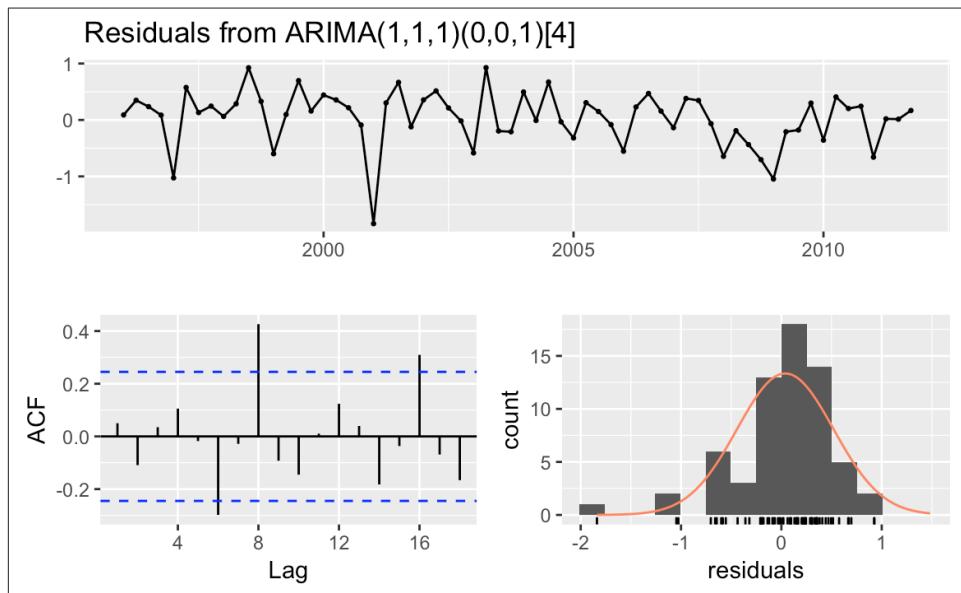


Figure 14-15. Residuals plots: problem model

The issues here are:

- The ACF shows significant autocorrelations between residuals.
- The p -values for the Ljung–Box statistics are small, indicating there is some pattern in the residuals (i.e., there is still information to be extracted from the data).
- The residuals appear asymmetrical.

These are basic diagnostics, but they are a good start. Find a good book on ARIMA modeling and perform the recommended diagnostic tests before concluding that your model is sound. Additional checks of the residuals could include:

- Tests for normality
- Quantile–quantile (Q–Q) plot
- Scatter plot against the fitted values

14.21 Making Forecasts from an ARIMA Model

Problem

You have an ARIMA model for your time series that you built with the `forecast` package. You want to forecast the next few observations in the series.

Solution

Save the model in an object, and then apply the `forecast` function to the object. This example saves the model from Recipe 14.19 and predicts the next eight observations:

```
m <- arima(euretail, order = c(0, 1, 3), seasonal = c(0, 1, 1),
             fixed = c(0, NA, NA, NA))
forecast(m)
#>           Point Forecast Lo 80 Hi 80 Lo 95 Hi 95
#> 2012 Q1          95.1  94.6  95.6  94.3  95.9
#> 2012 Q2          95.2  94.5  95.9  94.1  96.3
#> 2012 Q3          95.2  94.2  96.3  93.7  96.8
#> 2012 Q4          95.3  93.9  96.6  93.2  97.3
#> 2013 Q1          94.5  92.8  96.1  91.9  97.0
#> 2013 Q2          94.5  92.6  96.5  91.5  97.5
#> 2013 Q3          94.5  92.3  96.7  91.1  97.9
#> 2013 Q4          94.5  92.0  97.0  90.7  98.3
```

Discussion

The `forecast` function will calculate the next few observations and their standard errors according to the model. It returns a list with 10 elements. When we print the model, as we just did, `forecast` returns the time series points it is forecasting, the forecast, and two pairs of confidence bands: high/low 80% and high/low 95%.

If we want to extract out just the forecast, we can do that by assigning the results to an object, and then pulling out the list item named `mean`:

```
fc_m <- forecast(m)
fc_m$mean
#>      Qtr1 Qtr2 Qtr3 Qtr4
```

```
#> 2012 95.1 95.2 95.2 95.3  
#> 2013 94.5 94.5 94.5 94.5
```

The result is a `Time-Series` object containing the forecasts created by the `forecast` function.

14.22 Plotting a Forecast

Problem

You have created a time series forecast with the `forecast` package and you would like to plot it.

Solution

Time series models created with the `forecast` package have a plotting method that uses `ggplot2` to create graphs easily, as shown in [Figure 14-16](#):

```
fc_m <- forecast(m)  
autoplot(fc_m)
```

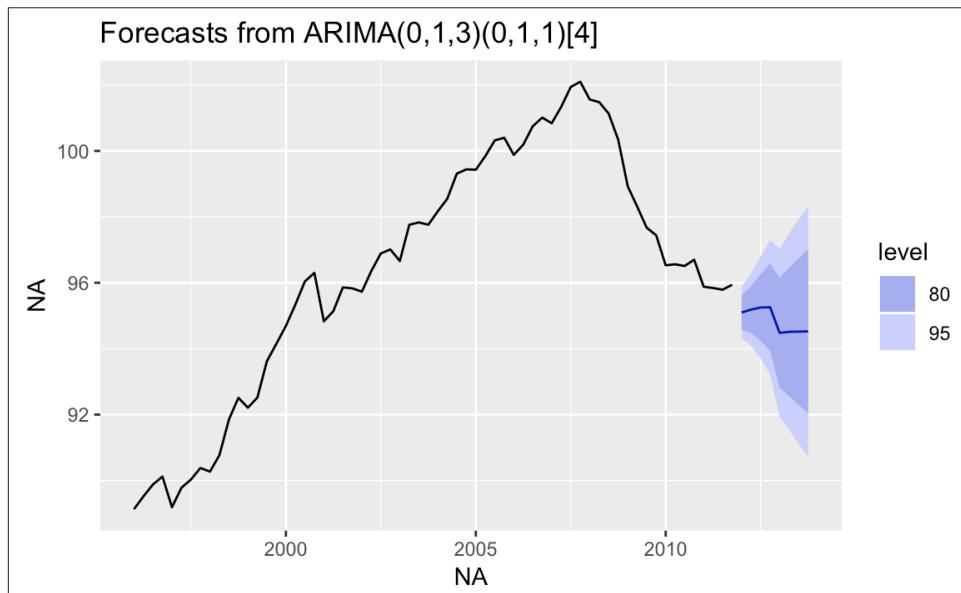


Figure 14-16. Forecast cone of uncertainty: default

Discussion

The `autoplot` function makes a very reasonable figure, as shown in [Figure 14-16](#). Since the resulting figure is a `ggplot` object, we can adjust the plotting parameters the

same way we would with any other `ggplot` object. Here we add labels and a title and change the theme, as shown in [Figure 14-17](#):

```
autoplot(fc_m) +  
  ylab("Euro Index") +  
  xlab("Year/Quarter") +  
  ggtitle("Forecasted Retail Index") +  
  theme_bw()
```

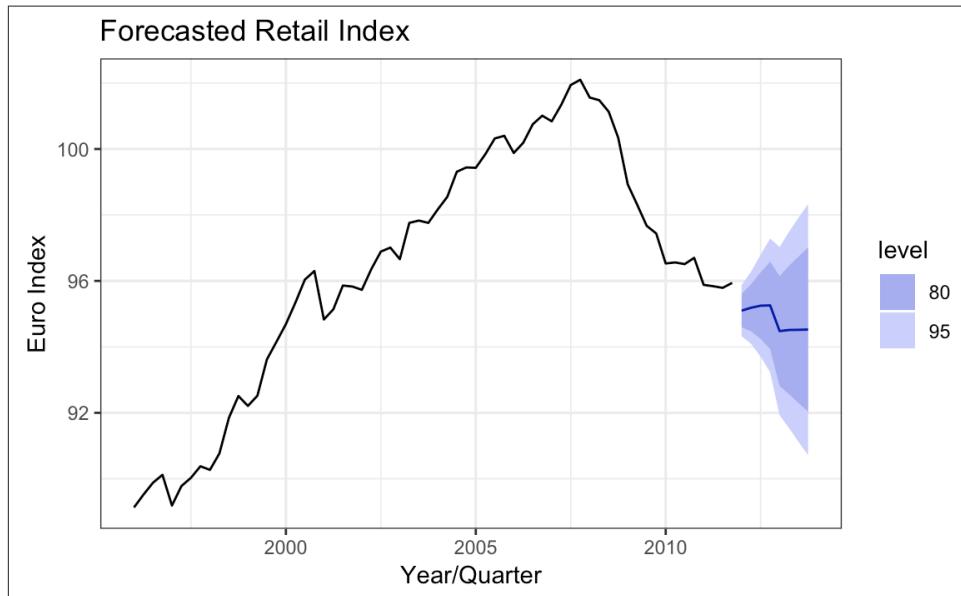


Figure 14-17. Forecast cone of uncertainty: labeled

See Also

See [Chapter 10](#) for more information on working with `ggplot` figures.

14.23 Testing for Mean Reversion

Problem

You want to know if your time series is mean-reverting (stationary).

Solution

A common test for mean reversion is the Augmented Dickey–Fuller (ADF) test, which is implemented by the `adf.test` function of the `tseries` package:

```
library(tseries)  
adf.test(ts)
```

The output from `adf.test` includes a p -value. Conventionally, if $p < 0.05$, the time series is likely mean-reverting, whereas a $p > 0.05$ provides no such evidence.

Discussion

When a time series is mean-reverting, it tends to return to its long-run average. It may wander off, but eventually it wanders back. If a time series is not mean-reverting, then it can wander away without ever returning to the mean.

Figure 14-18 appears to be wandering upward and not returning. The large p -value from `adf.test` confirms that it is not mean-reverting:

```
library(tseries)
library(fpp2)
autoplots(goog200)
adf.test(goog200)
#>
#> Augmented Dickey-Fuller Test
#>
#> data: goog200
#> Dickey-Fuller = -2, Lag order = 5, p-value = 0.7
#> alternative hypothesis: stationary
```

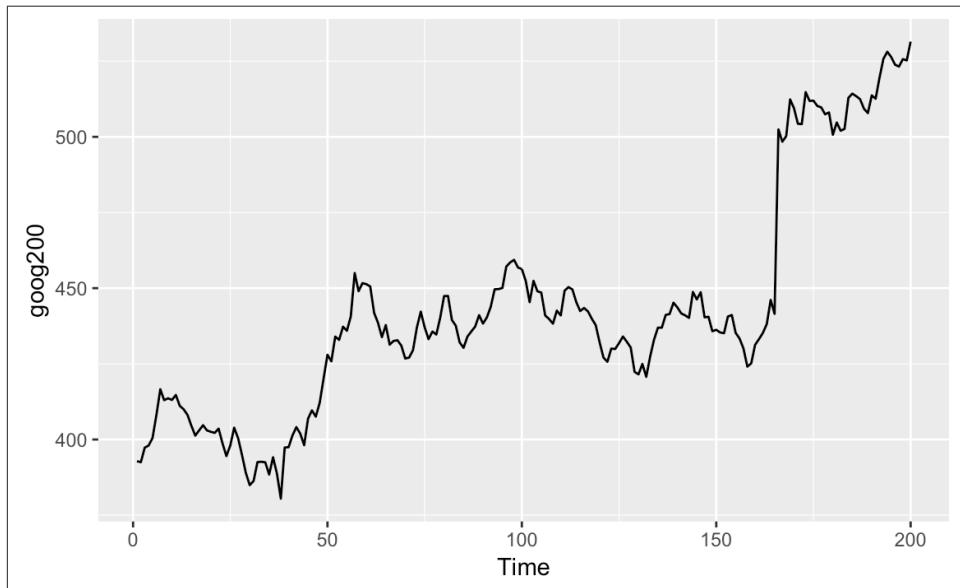


Figure 14-18. Time series without mean reversion

The time series in Figure 14-19, however, is just bouncing around its average value. The small p -value (0.01) confirms that it is mean-reverting:

```

autoplott(hsales)
adf.test(hsales)
#>
#> Augmented Dickey-Fuller Test
#>
#> data: hsales
#> Dickey-Fuller = -4, Lag order = 6, p-value = 0.01
#> alternative hypothesis: stationary

```

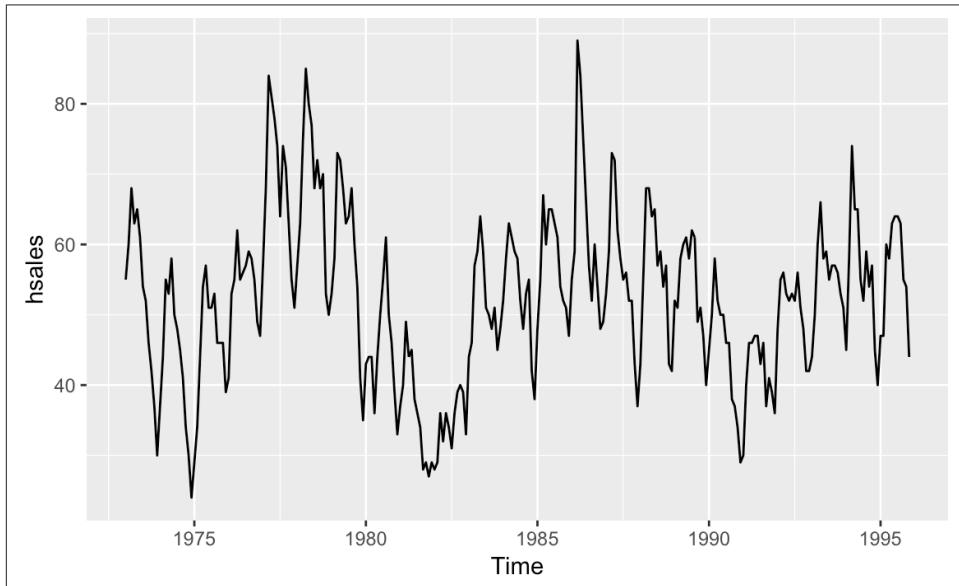


Figure 14-19. Time series with mean reversion

The example data here comes from the `fpp2` package and comprises all `Time-Series` object types. If your data were in a `zoo` or `xts` object, then you would need to call `coredata` to extract out the raw data from the object before passing it to `adf.test`:

```

library(xts)
data(sample_matrix)
xts_obj <- as.xts(sample_matrix, dateFormat = "Date")[, "Close"] # vector of data

adf.test(coredata(xts_obj))
#>
#> Augmented Dickey-Fuller Test
#>
#> data: coredata(xts_obj)
#> Dickey-Fuller = -3, Lag order = 5, p-value = 0.3
#> alternative hypothesis: stationary

```

The `adf.test` function massages your data before performing the ADF test. First it automatically detrends your data, and then it recenters the data, giving it a mean of zero.

If either detrending or recentering is undesirable for your application, use the `adfTest` function in the `fUnitRoots` package instead:

```
library(fUnitRoots)
adfTest(coredata(ts1), type = "nc")
```

With `type = "nc"`, the function neither detrends nor recenters your data. With `type = "c"`, the function recenters your data but does not detrend it.

Both the `adf.test` and `adfTest` functions let you specify a lag value that controls the exact statistic they calculate. These functions provide reasonable defaults, but serious users should study the textbook description of the ADF test to determine the appropriate lag for their application.

See Also

The `urca` and `CADFtest` packages also implement tests for a unit root, which is the test for mean reversion. Be careful when comparing the tests from several packages, however. Each package can make slightly different assumptions, which can lead to puzzling differences in the results.

14.24 Smoothing a Time Series

Problem

You have a noisy time series. You want to smooth the data to eliminate the noise.

Solution

The `KernSmooth` package contains functions for smoothing. Use the `dpline` function to select an initial bandwidth parameter, and then use the `locpoly` function to smooth the data:

```
library(KernSmooth)

gridsize <- length(y)
bw <- dpline(t, y, gridsize = gridsize)
lp <- locpoly(x = t, y = y, bandwidth = bw, gridsize = gridsize)
smooth <- lp$y
```

Here, `t` is the time variable and `y` is the time series.

Discussion

The `KernSmooth` package is a standard part of the R distribution. It includes the `locpoly` function, which constructs, around each data point, a polynomial that is

fitted to the nearby data points. These are called *local polynomials*. The local polynomials are strung together to create a smoothed version of the original data series.

The algorithm requires a bandwidth parameter to control the degree of smoothing. A small bandwidth means less smoothing, in which case the result follows the original data more closely. A large bandwidth means more smoothing, so the result contains less noise. The tricky part is choosing just the right bandwidth: not too small, not too large.

Fortunately, KernSmooth also includes the function `dpill` for estimating the appropriate bandwidth, and it works quite well. We recommend that you start with the `dpill` value and then experiment with values above and below that starting point. There is no magic formula here. You need to decide what level of smoothing works best in your application.

The following is an example of smoothing. We'll create some example data that is the sum of a simple sine wave and normally distributed "noise":

```
t <- seq(from = -10, to = 10, length.out = 201)
noise <- rnorm(201)
y <- sin(t) + noise
```

Both `dpill` and `locpoly` require a grid size—in other words, the number of points for which a local polynomial is constructed. We often use a grid size equal to the number of data points, which yields a fine resolution. The resulting time series is very smooth. You might use a smaller grid size if you want a coarser resolution or if you have a very large dataset:

```
library(KernSmooth)
gridsize <- length(y)
bw <- dpill(t, y, gridsize = gridsize)
```

The `locpoly` function performs the smoothing and returns a list. The `y` element of that list is the smoothed data:

```
lp <- locpoly(x = t, y = y, bandwidth = bw, gridsize = gridsize)
smooth <- lp$y

ggplot() +
  geom_line(aes(x = t, y = y)) +
  geom_line(aes(x = t, y = smooth), linetype = 2)
```

In Figure 14-20, the smoothed data is shown as a dashed line, while the solid line is our original example data. The figure demonstrates that `locpoly` did an excellent job of extracting the original sine wave.

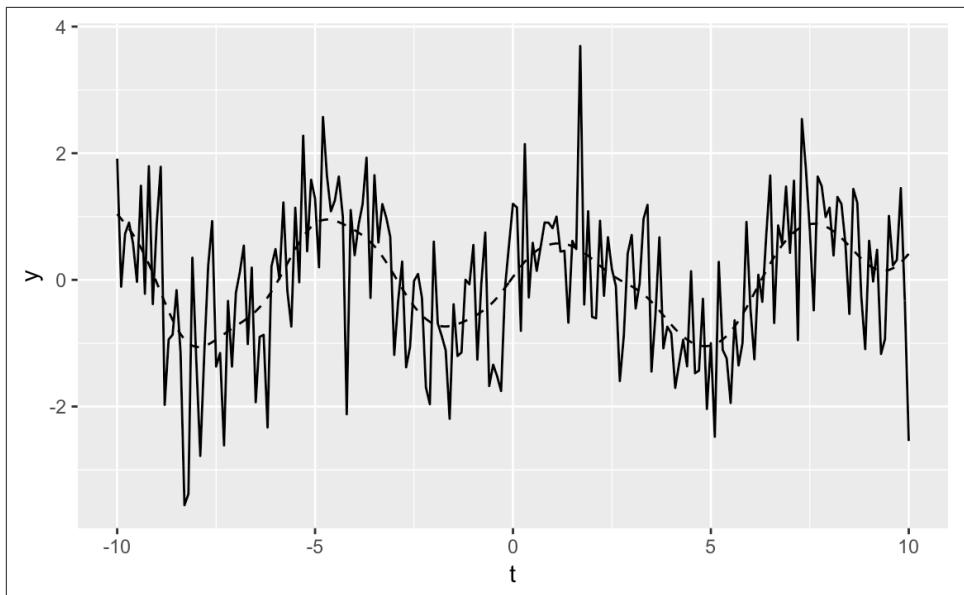


Figure 14-20. Example time series plot

See Also

The `ksmooth`, `lowess`, and `HoltWinters` functions in the base distribution can also perform smoothing. The `expsmooth` package implements exponential smoothing.

Simple Programming

R lets you accomplish a lot without knowing anything about programming. Programming opens the door to accomplishing more, however, and most serious users eventually perform some level of programming, starting simply and possibly becoming quite proficient. While this is not a programming book, this chapter lays out some programming recipes that R users typically find useful to begin their journey.

If you are already familiar with programming and programming languages, a few notes here may help you quickly adapt. (If these terms are unfamiliar to you, you can skip this section.) Here are some technical details of R to be aware of:

Typeless variables

Variables in R do not have a fixed type, such as integer or character, unlike in typed languages such as C and Java. A variable could contain a number one moment and a data frame the next.

Return values

All functions return a value. Normally, a function returns the value of the last expression in its body. You can also use `return(expr)` anywhere within the body.

Call-by-value parameters

Function parameters are “call by value”—in other words, parameters are strictly local variables, and changes to those variables do not affect the caller’s value.

Local variables

You create a local variable simply by assigning a value to it. Explicit declaration is not required. When the function exits, local variables are lost.

Global variables

Global variables are held in the user’s workspace. Within a function you can change a global variable by using the <- assignment operator, but this is not encouraged.

Conditional execution

The R syntax includes an `if` statement. See `help(Control)` for details.

Loops

The R syntax also includes `for` loops, `while` loops, and `repeat` loops. For details, see `help(Control)`.

Case or switch statements

A special function called `switch` provides a basic case statement. The semantics may strike you as odd, however. See `help(switch)` for details.

Lazy evaluation

R does not immediately evaluate function arguments when the function is called. Rather, it waits until the argument is actually used within the function, then evaluates it. This gives the language an especially rich and powerful semantics. Most of the time, it’s not noticeable, but occasionally it results in situations that are baffling to programmers familiar only with “eager” evaluation, where arguments are evaluated when the function is called.

Functional semantics

Functions are “first-class citizens” and can be treated like other objects: assigned to variables, passed to functions, printed, inspected, and so forth.

Object orientation

R supports object-oriented programming. In fact, there are several different paradigms for object orientation, which is a blessing if you enjoy having a choice and baffling if you don’t.

15.1 Choosing Between Two Alternatives: `if/else`

Problem

You want to write a *conditional branch* that will choose between two paths based on a simple test.

Solution

An `if` block can implement conditional logic by testing a simple condition:

```
if (condition) {  
  ## do this if condition is TRUE
```

```
    } else {
        ## do this if condition is FALSE
    }
```

Notice the parentheses around the condition, which are required, and the curly braces around the subsequent two blocks of code.

Discussion

The `if` structure lets you choose between two alternative code paths by testing some condition, such as `x == 0` or `y > 1`, and then following one path or the other accordingly. This `if`, for example, checks for negative numbers before calculating a square root:

```
if (x >= 0) {
    print(sqrt(x))           # do this if x >= 0
} else {
    print("negative number")  # do this otherwise
}
```

You can chain a series of `if/else` structures to make a series of decisions. Let's suppose we want a value to be capped at 0 (no negative values) and capped at 1. We could code that as follows:

```
x <- -0.3

if (x < 0) {
    x <- 0
} else if (x > 1) {
    x <- 1
}

print(x)
#> [1] 0
```

It is important that the conditional test (the expression after `if`) is a *simple* test; that is, it must return a single, logical value of either TRUE or FALSE. A common problem is mistakenly using a *vector* of logical values, as in this example:

```
x <- c(-2, -1, 0, 1, 2)

if (x < 0) {
    print("values are negative")
}
#> Warning in if (x < 0) {: the condition has length > 1 and only the first
#> element will be used
#> [1] "values are negative"
```

The problem arises because `x < 0` is ambiguous when `x` is a vector: are you testing for *all* values being negative or *some* values being negative? R provides the helper

functions `all` and `any` to address the situation. They take a vector of logical values and reduce them to one, single value:

```
x <- c(-2, -1, 0, 1, 2)

if (all(x < 0)) {
  print("all are negative")
}

if (any(x < 0)) {
  print("some are negative")
}
#> [1] "some are negative"
```

See Also

The `if` structure presented here is intended for programming. There is also a function called `ifelse` that implements a vectorized `if/else` structure, useful for transforming entire vectors. See `help(ifelse)`.

15.2 Iterating with a Loop

Problem

You want to iterate over the elements of a vector or list.

Solution

A common iteration technique uses the `for` structure. If `v` is a vector or list, this `for` loop selects each element of `v` one by one, assigns the element to `x`, and does something with it:

```
for (x in v) {
  # do something with x
}
```

Discussion

Programmers from C and Python will recognize `for` loops. They are less common in R but still occasionally useful.

For illustration, this `for` loop prints the first five integers and their squares. It sets `x` to 1, 2, 3, 4, and 5 successively, executing the body of the loop each time:

```
for (x in 1:5) {
  cat(x, x^2, "\n")
}
#> 1 1
#> 2 4
```

```
#> 3 9  
#> 4 16  
#> 5 25
```

We can also iterate over the *subscripts* of a vector or list, which is useful for updating the data in place. Here, we initialize `v` with the vector `1:5`, then update its elements by squaring each one:

```
v <- 1:5  
for (i in 1:5) {  
  v[[i]] <- v[[i]] ^ 2  
}  
print(v)  
#> [1] 1 4 9 16 25
```

But, frankly, this also illustrates one reason why loops are less common in R than in other programming languages. The vectorized operations of R are fast and easy, often eliminating the need for looping altogether. Here is the vectorized version of the previous example:

```
v <- 1:5  
v <- v^2  
print(v)  
#> [1] 1 4 9 16 25
```

See Also

Another reason loops are rare is that `map` and similar functions can process entire vectors and lists at once, usually more quickly and easily than a loop. See [Recipe 6.1](#) for details on using the `purrr` package to apply functions to lists.

15.3 Defining a Function

Problem

You want to define a new R function.

Solution

Create the function by using the `function` keyword followed by a list of parameter names and then the function body:

```
name <- function(param1, ..., paramN) {  
  expr1  
  .  
  .  
  .  
  exprM  
}
```

Put parentheses around the parameter names. Put curly braces around the function body, which is a sequence of one or more expressions. R will evaluate each expression in order and return the value of the last one, denoted here as expr_M .

Discussion

Function definitions are how you tell R, “Here’s how to calculate *this*.” For example, R does not have a built-in function for calculating the coefficient of variation, but we can create such a function, calling it `cv`:

```
cv <- function(x) {  
  sd(x) / mean(x)  
}
```

This function has one parameter, `x`, and the body of the function is `sd(x) / mean(x)`.

When we call the function with an argument, R will set the parameter `x` to that value, then evaluate the body of the function:

```
cv(1:10)      # Set x = 1:10 and evaluate sd(x)/mean(x)  
#> [1] 0.550482
```

Note that the parameter `x` is distinct from any other variable called `x`. If you have a global variable `x` in your workspace, for example, that `x` is distinct from this `x` and won’t be affected by `cv`. Furthermore, the parameter `x` exists only while the `cv` function is executing and disappears after that.

A function can have more than one argument. This function has two arguments, both integers, and implements Euclid’s algorithm for computing their greatest common divisor:

```
gcd <- function(a, b) {  
  if (b == 0) {  
    a                  # Return a to caller  
  } else {  
    gcd(b, a %% b)    # Recursively call ourselves  
  }  
}  
  
# What's the greatest common denominator of 14 and 21?  
gcd(14, 21)  
#> [1] 7
```

(This function definition is *recursive* because it calls itself when `b` is nonzero.)

Normally, the function returns the value of the last expression in the function body. You can choose to return a value earlier, however, by writing `return(expr)`, forcing the function to stop and immediately return `expr` to the caller. We can illustrate this by coding `gcd` in a subtly different way using an explicit `return`:

```
gcd <- function(a, b) {  
  if (b == 0) {  
    return(a)    # Stop and return a  
  }  
  gcd(b, a %% b)  
}
```

When parameter `b` is 0, `gcd` executes `return(a)`, returning that value immediately to the caller.

See Also

Functions are a central component of R programming, so they are covered well in books such as *R for Data Science* by Hadley Wickham and Garrett Grolemund (O'Reilly) and *The Art of R Programming* by Norman Matloff (No Starch Press).

15.4 Creating a Local Variable

Problem

You want to create a variable that is *local* to a function—that is, a variable that is created inside the function, used inside the function, and removed when the function is done.

Solution

Inside the function, simply assign a value to the name. The name automatically becomes a local variable and will be removed when the function finishes.

Discussion

This function will map a vector, `x`, into the unit interval. It requires two intermediate values, `low` and `high`:

```
unitInt <- function(x) {  
  low <- min(x)  
  high <- max(x)  
  (x - low) / (high - low)  
}
```

The `low` and `high` values are automatically created by the assignment statements. Because the assignments occur within the function body, the variables are *local* to the function. That brings two important advantages.

First, the local variables named `low` and `high` are distinct from any global variables named `low` and `high` in your workspace. Because they are distinct, there is no “collision”: changes to the local variables do not change the global variables.

Second, local variables disappear when the function is done. That prevents clutter and automatically frees the space they used.

15.5 Choosing Between Multiple Alternatives: switch

Problem

A variable can take on several different values. You want your program to handle each case separately, according to the value.

Solution

The `switch` function will branch according to a value, letting you select how you handle each case.

Discussion

The first argument to `switch` is a value for R to consider. The remaining arguments show how to handle each possible value. For example, this call to `switch` considers the value of `who`, then returns one of three possible results:

```
hair_type = switch(who,
  Moe = "long",
  Larry = "fuzzy",
  Curly = "none")
```

Notice that each expression after the initial `who` is labeled with a possible value for `who`. If `who` is `Moe`, then the `switch` returns `"long"`; if it is `Larry`, the `switch` returns `"fuzzy"`; if it's `Curly`, it returns `"none"`.

Very often, you cannot anticipate all possible values to be considered, so `switch` lets you define a default for the situation where no label matches. Simply put the default last with no label. This `switch`, for example, will translate the contents of `s` from `"one"`, `"two"`, or `"three"` into the corresponding integer. It returns `NA` for any other value:

```
num <- switch(s,
  one = 1,
  two = 2,
  three = 3,
  NA)
```

An annoying quirk of `switch` arises when the labels are integers. This won't do what you expect, for example:

```
switch(i,           # Does not work the way you expect
  10 = "ten",
  20 = "twenty",
```

```
30 = "thirty",
"other")
```

But there is a workaround—convert the integer to a character string, then use character strings for the labels:

```
switch(as.character(i),
      "10" = "ten",
      "20" = "twenty",
      "30" = "thirty",
      "other")
```

See Also

See `help(switch)` for more details.

This sort of feature is quite common in other programming languages, where it's usually called a *switch* or *case* statement.

The `switch` function works only with scalars. Switching on the contents of a data frame is more complicated. See the function `case_when` in the `dplyr` package for a powerful mechanism to handle that situation.

15.6 Defining Defaults for Function Parameters

Problem

You want to define default parameters for a function—that is, values to use when the caller does not provide explicit arguments.

Solution

R lets you set default values for parameters by including them in the function definition:

```
my_fun <- function(param = default_value) {
  ...
}
```

Discussion

Let's create a toy function that greets someone by name:

```
greet <- function(name) {
  cat("Hello, ", name, "\n")
}

greet("Fred")
#> Hello, Fred
```

If we call `greet` without a `name` argument, we get this error:

```
greet()  
#> Error in cat("Hello,", name, "\n") :  
#>   argument "name" is missing, with no default
```

We can change the function definition, however, to define a default name. In this case, we'll default to the generic name `world`:

```
greet <- function(name = "world") {  
  cat("Hello,", name, "\n")  
}
```

Now if we omit the argument, R supplies a default:

```
greet()  
#> Hello, world
```

This mechanism for defaults is handy. Nonetheless, we recommend using it judiciously. We've seen too many cases where the function *creator* defined defaults and the function *caller* accepted the defaults without much thought, leading to questionable results. For example, if you are using the k -nearest neighbors algorithm, the choice of k is critical and providing a default makes no sense. Sometimes it's better to force the caller to make a choice.

15.7 Signaling Errors

Problem

When your code encounters a serious problem, you want to halt and alert the user.

Solution

Call the `stop` function, which will print your message and terminate all processing.

Discussion

It is critical to halt processing when your code encounters fatal errors, such as this check that an account still has a positive balance:

```
if (balance < 0) {  
  stop("Funds exhausted.")  
}
```

This call to `stop` would display the message, terminate processing, and put the user back at the console prompt:

```
#> Error in eval(expr, envir, enclos): Funds exhausted
```

Problems arise for all sorts of reasons: bad data, user error, network failures, and bugs in code, to name a few. The list is endless. It is important that you anticipate potential problems and code appropriately:

Detect

At a minimum, detect possible errors. Halt if further processing is impossible. Undetected errors are a major source of program failures.

Report

If you must halt, give users a reasonable explanation of why. That will help them diagnose and fix the problem.

Recover

In some cases, the code may be able to correct the situation itself and continue. We recommend, however, warning the user that your code encountered a problem and corrected it.

Error handling is part of *defensive programming*, the practice of making your code robust.

See Also

An alternative to `stop` is the `warning` function, which prints its message and continues without halting. Be sure, however, that it is actually reasonable to continue.

15.8 Protecting Against Errors

Problem

You anticipate the possibility of fatal errors, and you want to handle them rather than halt altogether.

Solution

Use the `possibly` function to “wrap” the problematic code. It will trap errors and let you respond to them.

Discussion

The `purrr` package contains a function called `possibly`, which takes two parameters. The first parameter is a function, and `possibly` will protect against failures in that function. The second parameter is a value called `otherwise`.

A concrete example is useful here. The `read.csv` function tries to read a file, but it simply halts if the file does not exist. That could be undesirable. We might want to recover and continue instead.

We can “wrap” the `read.csv` function in a protective layer this way:

```
library(purrr)
safe_read <- possibly(read.csv, otherwise=NULL)
```

It may seem strange, but `possibly` returns a *new function*. The new function, called `safe_read` here, behaves exactly like the old function, `read.csv`, but with one very important difference. When `read.csv` would fail and halt, `safe_read` will instead return the `otherwise` value (`NULL`) and let you continue. (If `read.csv` succeeds, you get its usual result: a data frame.)

You could use `safe_read` like this to handle optional files:

```
details = safe_read("details.csv")      # Try to read details.csv file
if (is.null(details)) {                 # NULL means read.csv failed
  cat("Details are not available\n")
} else {
  print(details)                      # We got the contents!
}
```

If the `details.csv` file exists, `safe_read` returns the contents and this code will print them. If it does not exist, then `read.csv` fails, `safe_read` returns `NULL`, and this code prints a message.

The `otherwise` value in this case is `NULL`, but it can be anything. It could be a data frame, for example, which provides a default. In that case, when the `details.csv` file is unavailable, `safe_read` would return that default.

See Also

The `purrr` package contains other functions for protecting against errors. Check out the `safely` and `quietly` functions.

If you need even higher-powered tools, use `help(tryCatch)` to see the mechanism behind `possibly`, which has sophisticated bells and whistles for handling both errors and warnings. It mirrors the familiar `try/catch` paradigm of other programming languages.

15.9 Creating an Anonymous Function

Problem

You are using tidyverse functions such as `map` or `discard` that require a function. You want a shortcut for easily defining the required function.

Solution

Use the `function` keyword to define a function with parameters and a body, but instead of giving the function a name, simply use its definition inline.

Discussion

It may seem strange to create a function with no name, but it can be a handy convenience.

In [Recipe 15.3](#), we defined a function, `is_na_or_null`, and used it to remove NA and NULL elements from a list:

```
is_na_or_null <- function(x) {  
  is.na(x) || is.null(x)  
}  
  
lst %>%  
  discard(is_na_or_null)
```

Sometimes, writing a tiny, one-off function such as `is_na_or_null` is annoying. You can avoid that hassle by using the function definition directly, not giving it a name:

```
lst %>%  
  discard(function(x) is.na(x) || is.null(x))
```

This kind of function is called an *anonymous function*, for the obvious reason that it has no name.

See Also

Function definitions are described in [Recipe 15.3](#).

15.10 Creating a Collection of Reusable Functions

Problem

You want to reuse one or more functions across several scripts.

Solution

Save the functions in a local file, say `myLibrary.R`, then use the `source` function to load those functions into your script:

```
source("myLibrary.R")
```

Discussion

Quite often, you will write functions that are useful in several scripts. For example, you could have one function that loads, checks, and cleans your data; now you want to reuse that function in every script that needs the data.

Most beginners simply cut and paste the reusable function into each script, duplicating the code. That creates a serious problem. What if you discover a bug in that duplicated code? Or what if you must change the code to accommodate new circumstances? You're forced to hunt down every copy and make the identical change everywhere, an annoying and error-prone process.

Instead, create a file, say *myLibrary.R*, and save the function definition there. The file contents could look like this:

```
loadMyData <- function() {  
  # code for data loading, checking, and cleaning here  
}
```

Then, inside each script, use the `source` function to read the code from the file:

```
source("myLibrary.R")
```

When you run the script, the `source` function reads the indicated file, just as if you'd typed the file contents at that location in the script. It's better than cutting and pasting because you've isolated the function's definition into one known place.



This example has only one function in the sourced file, but the file can contain multiple functions, of course. We suggest gathering related functions into their own file, creating a group of related, reusable functions.

See Also

This recipe is a very simple method for reusing code, appropriate for small projects. A more powerful approach is to create your own R package of functions, which is especially useful for collaborating with other people. Package creation is a large topic, but getting started is pretty easy. We suggest the excellent book *R Packages* by Hadley Wickham (O'Reilly), available in printed form or [online](#).

15.11 Automatically Reindenting Code

Problem

You want to reformat your code so that it lines up nicely and is indented consistently.

Solution

To consistently indent a block of code, highlight the text in RStudio, then press Ctrl-I (Windows or Linux) or Cmd-I (Mac).

Discussion

One of the many features of the RStudio IDE is that it helps with routine code maintenance, such as reformatting. When you're editing code it's easy to end up with indentation that is inconsistent and a little confusing. The IDE can fix that.

Take the following code, for example:

```
for (i in 1:5) {  
  if (i >= 3) {  
    print(i**2)  
  } else {  
    print(i * 3)  
  }  
}
```

While that's valid code, it can be tricky to read because of the odd indentation. If we highlight the text in the RStudio IDE and press Ctrl-I (or Cmd-I on Mac), then our code gets consistent indentation:

```
for (i in 1:5) {  
  if (i >= 3) {  
    print(i**2)  
  }  
  else {  
    print(i * 3)  
  }  
}
```

See Also

RStudio has several helpful features for code editing. You can access cheat sheets by clicking Help → Cheatsheets or by going directly to <https://www.rstudio.com/resources/cheatsheets/>.

R Markdown and Publishing

While R by itself is an incredibly powerful tool for data analysis and visualization, almost all of us, after we do analysis, will need to communicate the results to others. We may do that with published papers, blog posts, PowerPoint presentations, or books. R Markdown is the tool that helps us go from R analysis and visualization all the way to publishable documents.

R Markdown is a package (as well as an ecosystem of tools) that allows us to add R code to a plain-text file with some Markdown formatting. The document can then be rendered into many different output formats, including PDF, HTML, Microsoft Word, and Microsoft PowerPoint. At rendering, also called *knitting*, the R code is run and the resulting output and figures are placed in the final document.

In this chapter we'll give you recipes to get you started creating R Markdown documents. After you go through these recipes, one of the best ways to learn more about R Markdown is by looking at the source files and final output of other people's R Markdown work. The book you are reading was itself written in R Markdown. You can see the source to this book on [GitHub](#).

In addition, Yihui Xie, J. J. Allaire, and Garrett Grolemund have written *R Markdown: The Definitive Guide* (Chapman & Hall/CRC) and also made the source R Markdown available on [GitHub](#).

Many other books written with R Markdown have been made [freely available online](#).

We mentioned that R Markdown is an ecosystem as well as a package. There are specialized packages to extend R Markdown for blogging (`blogdown`), for books (`bookdown`), and for making gridded dashboards (`flexdashboard`). The initial package in the ecosystem is called `knitr`, and we still call the process of turning R Markdown into a final format “knitting” the document. The R Markdown ecosystem supports many output formats, and covering them all would be unreasonable. In this book

we'll stick primarily to four common output formats: HTML, LaTeX, Microsoft Word, and Microsoft PowerPoint.

The RStudio IDE contains many helpful features for creating and editing R Markdown documents. While we'll make use of those features in the following recipes, R Markdown is not dependent on RStudio in order to be useful. It's possible to edit plain-text R Markdown files with your favorite text editor and then knit the document using R's command-line interface. However, the RStudio tools are so helpful that we'll illustrate them extensively.

16.1 Creating a New Document

Problem

You want to create a new R Markdown document to tell your data story.

Solution

The easiest way to create a new R Markdown document is using the File → New File → R Markdown... menu choice in the RStudio IDE (see [Figure 16-1](#)).

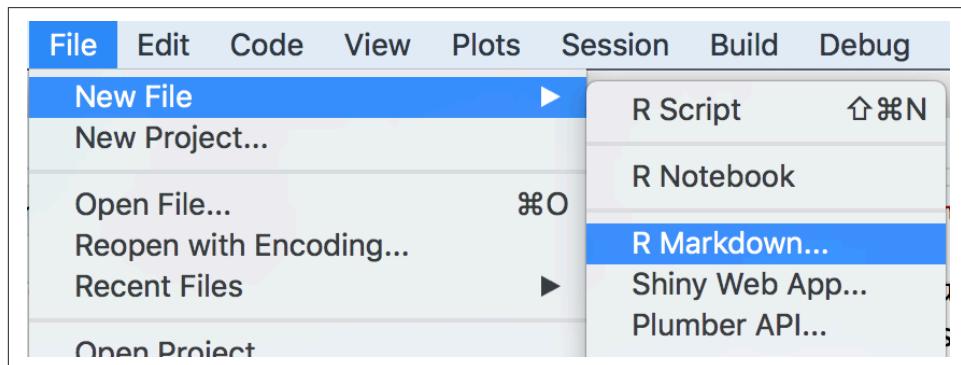


Figure 16-1. Creating a new R Markdown document

Selecting “R Markdown...” will lead you to the New R Markdown dialog, where you can choose the type of output document you would like to create (see [Figure 16-2](#)). The default option is HTML, which is a good choice if you want to publish your work online or in an email, or if you haven't made up your mind yet about how you'd like to output your final document. Changing to a different format later is typically as easy as chaining one line of text in the document, or a few clicks in the IDE.

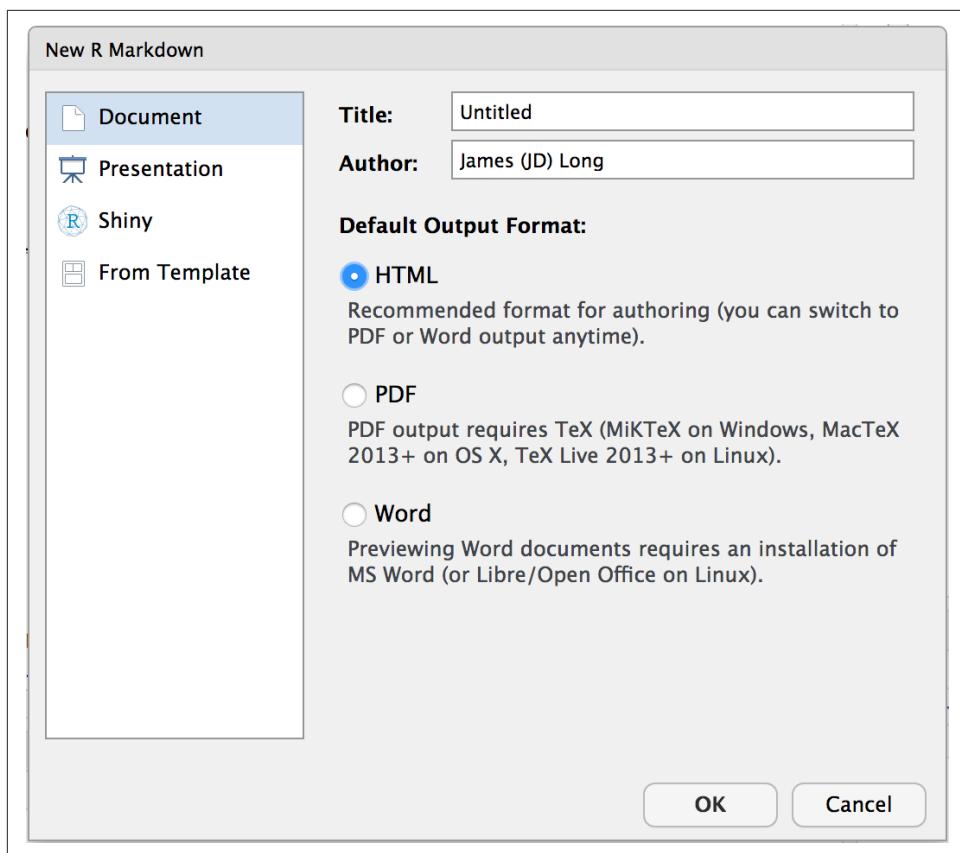
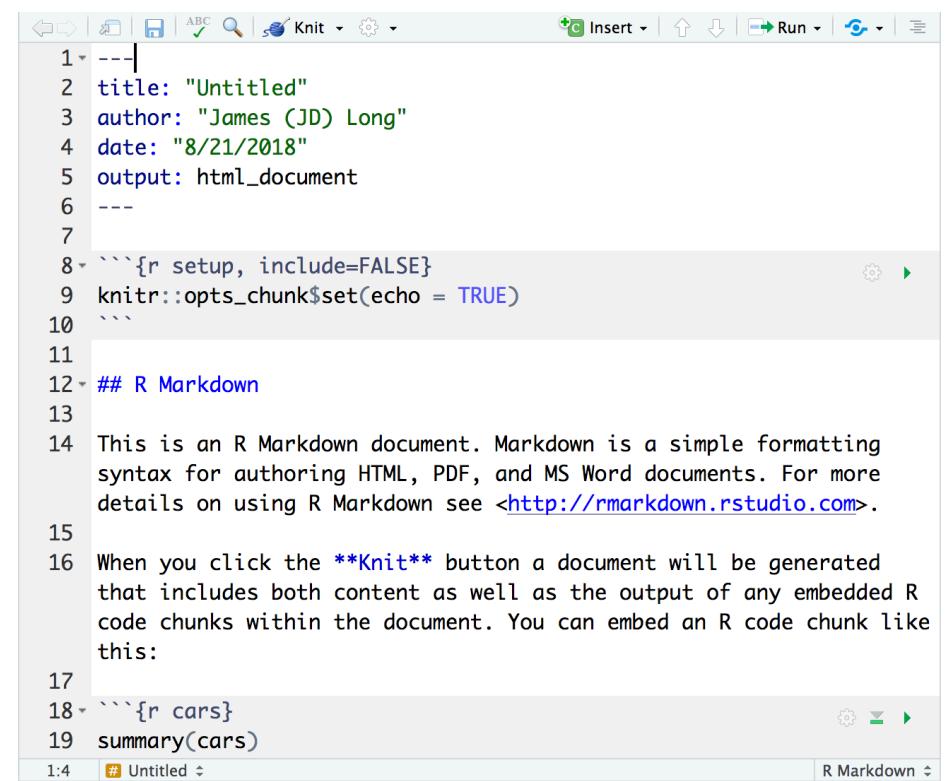


Figure 16-2. New R Markdown document options

After you make your selection and click OK, you'll get an R Markdown template with some metadata and example text (see [Figure 16-3](#)).

A screenshot of the RStudio IDE interface. The top menu bar includes options like ABC, Knit, Insert, Run, and Help. The main code editor area contains the following R Markdown code:

```
1 ---|  
2 title: "Untitled"  
3 author: "James (JD) Long"  
4 date: "8/21/2018"  
5 output: html_document  
6 ---  
7  
8 `r setup, include=FALSE}  
9 knitr::opts_chunk$set(echo = TRUE)  
10 ````  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.  
15  
16 When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:  
17  
18 `r cars}  
19 summary(cars)  
1:4 # Untitled ▾ R Markdown ▾
```

The status bar at the bottom shows the file is titled "# Untitled".

Figure 16-3. New R Markdown document

Discussion

R Markdown documents are plain-text files. The shortcut just outlined is the fastest way to get a template for creating a new R Markdown text document. Once you have the template you can edit the text, alter the R code, and change anything you want. The other recipes in this chapter deal with the types of things you will likely want to do in your R Markdown document, but if you just want to see what some output looks like, click on the Knit button in the RStudio IDE and your R Markdown document will be rendered into your desired output format.

16.2 Adding a Title, Author, or Date

Problem

You want to alter the title, author, or date of your document.

Solution

At the top of an R Markdown document is a block of specially formatted text that starts and ends with `---`. This block contains important metadata about your document. In this block, you can set the title, author, and date:

```
---
```

```
title: "Your Title Here"
author: "Your Name Here"
date: "12/31/9999"
output: html_document
```

```
---
```

You can also set the output format (e.g., `output: html_document`). We'll discuss the different output formats later in the recipes that cover specific formats.

Discussion

When you knit your R Markdown document to create your output, R will run each chunk, create Markdown (not R Markdown) for each chunk's output, and pass the full Markdown document to Pandoc. Pandoc is the software that creates your final output document from the intermediate Markdown. Most of the time, you don't even need to think about the steps unless you're having a problem knitting your document.

The text at the top of your R Markdown document between the `---` marks is in a format called YAML (Yet Another Markup Language). This chunk is used to pass metadata to the Pandoc software that builds your output document. The fields `title`, `author`, and `date` are read by Pandoc and inserted at the top of most output document formats.

The way these values are formatted and inserted into the output document is a function of the template used for output. The default templates for HTML, PDF, and Microsoft Word each format the `title`, `author`, and `date` fields similarly (see [Figure 16-4](#)).



Figure 16-4. Header illustration

You can add other key/value pairs into the YAML header, but if your template is not configured to use these values, they are ignored.

See Also

For information on creating your own templates, see Chapter 17, “Document Templates,” in *R Markdown: The Definitive Guide*.

16.3 Formatting Document Text

Problem

You want to format the text of your document, such as putting text into italics or bold.

Solution

The body of an R Markdown document is plain text and allows formatting using Markdown notation. You’ll likely want to add formatting, such as making text **bold** or *italic*. You’ll also want to add section headers, lists, and tables, which will be covered in later recipes. All of these options can be accomplished through Markdown.

Table 16-1 shows a brief summary of some of the most common formatting syntax.

Table 16-1. Common Markdown formatting syntax

Markdown	Output
plain text	plain text
italics	<i>italics</i>
bold	bold
`code`	code
sub~script~	sub _{script}
super^script^	super ^{script}
~~strikethrough~~	strikethrough
endash: --	endash: –
emdash: ---	emdash: —

See Also

RStudio publishes a [handy reference sheet](#).

See also recipes for inserting various structures, such as [Recipe 16.4](#), [Recipe 16.5](#), and [Recipe 16.9](#).

16.4 Inserting Document Headings

Problem

Your R Markdown document needs section headings.

Solution

You can insert section headings by starting a line with the # (hash) character. Use one hash character for the top level, two for the second level, and so on:

```
# Level 1 Heading  
## Level 2 Heading  
### Level 3 Heading  
#### Level 4 Heading  
##### Level 5 Heading  
##### Level 6 Heading
```

Discussion

Markdown and HTML both support up to six heading levels, so that's what's supported in R Markdown. In R Markdown (and Markdown in general) the formatting does not include specific font details; it communicates only what formatting class to apply to text. The specifics of each class are defined by the output format and the template used by each output format.

16.5 Inserting a List

Problem

You want to include a bulleted or numbered list in your document.

Solution

To create a bulleted list, start each line with an asterisk (*) like so:

```
* first item  
* second item  
* third item
```

To create a numbered list, start each line with 1. as follows:

```
1. first item  
1. second item  
1. third item
```

R Markdown will replace the 1. prefixes with the sequence 1., 2., 3., and so on.

The rules for lists are a bit strict:

- There must be a blank line *before* the list.
- There must be a blank line *after* the list.
- There must be a space character after the leading asterisk.

Discussion

The syntax for lists is simple, but watch out for the rules given in the Solution. If you violate even one, the output will be gobbledegook.

An important feature of lists is that they allow sublists. This bulleted list has three subitems:

```
* first item
  * first subitem
  * second subitem
  * third subitem
* second item
```

which produces this output:

- first item
 - first subitem
 - second subitem
 - third subitem
- second item

Again, there is an important rule: the sublists must be indented by two, three, or four spaces relative to the level above. No more, no less—otherwise, chaos will ensue.

The Solution recommends using the prefix 1. to identify numbered lists. You can also use a. and i., which will produce lowercase letters and roman numeral sequences, respectively. That's handy for formatting sublists:

```
1. first item
1. second item
  a. subitem 1
  a. subitem 2
    i. sub-subitem 1
    i. sub-subitem 2
  a. subitem 2
1. third item
```

This produces:

1. first item
2. second item
 - a. subitem 1
 - b. subitem 2
 - i. sub-subitem 1
 - ii. sub-subitem 2
 - c. subitem 2
3. third item

See Also

The syntax for lists is more flexible and feature-laden than described here. See the reference material for details, such as the [Pandoc Markdown guide](#).

16.6 Showing Output from R Code

Problem

You want to execute some R code and show the results in the output document.

Solution

You can insert R code in an R Markdown document. It will be executed and the output included in the final document.

There are two ways to insert the code. For small bits of code, include them inline between two tick marks (` `), as in:

The square root of pi is `r sqrt(pi)`.

which results in this output:

The square root of pi is 1.772.

For larger blocks of code, define a *code chunk* by placing the block between matching triple tick marks (```).

```
```{r}
code block goes here
```
```

Note the {r} after the first triple tick marks: this alerts R Markdown that we want it to execute the code.

Discussion

Embedding R code into your document is the most powerful feature of R Markdown. In fact, without that feature, R Markdown would just be plain old Markdown.

Inline R, described first in the Solution, is useful for pulling in small bits of information directly into the text of a report—information such as dates, times, or the results of small calculations.

Code chunks are for doing the heavy lifting. By default, the code chunk is shown in the text, and the results are displayed directly under the code. The results are preceded by a prefix, which defaults to a double hash tag: `##`.

If we had this code chunk in a source R Markdown document:

```
```{r}
sqrt(pi)
sqrt(1:5)
...``
```

it would produce this output:

```
sqrt(pi)
[1] 1.77
sqrt(1:5)
[1] 1.00 1.41 1.73 2.00 2.24
```

Conveniently, having the results preceded by the `##` allows the reader to paste the code and results into their own R session and execute the code. R will ignore the results because they look like comments.



The `{r}` after the tick marks is important because R Markdown allows code blocks from other languages, too, such as Python or SQL. If you work in a multilanguage environment, this is a very powerful feature. See the R Markdown documentation for details.

## See Also

See [Recipe 16.7](#) for controlling what's shown in the output.

For details on the available language engines, see “Other language engines” in [\*R Markdown: The Definitive Guide\*](#).

# 16.7 Controlling Which Code and Results Are Shown

## Problem

Your document contains chunks of R code. You want to control what's shown in the final document: only the results, only the code, or neither.

## Solution

Code blocks support several options that control what appears in the final document. Set the options at the top of the block. For example, this block has echo set to FALSE:

```
```{r echo=FALSE}
# . . . code here will not appear in output . . .
````
```

See the Discussion for a table of available options.

## Discussion

There are many display options, such as echo, which controls whether the code itself appears in the final output, and eval, which controls whether or not the code is evaluated (executed).

A few of the most popular options are listed in [Table 16-2](#).

*Table 16-2. Options that control what's shown in the final document*

| Chunk option    | Executes code | Shows code | Shows output text | Shows figures |
|-----------------|---------------|------------|-------------------|---------------|
| results='hide'  | X             | X          |                   | X             |
| include=FALSE   | X             |            |                   |               |
| echo=FALSE      | X             |            | X                 | X             |
| fig.show='hide' | X             | X          | X                 |               |
| eval=FALSE      |               | X          |                   |               |

You can mix and match combinations of options to get the results you're after. Some common use cases are:

- You want the code's output to appear, but not the code itself: `echo=FALSE`.
- You want the code to appear, but not be executed: `eval=FALSE`.
- You want to execute the code for its side effects (e.g., loading packages or loading data), but neither the code nor any incidental output should appear: `include=FALSE`.

We often use `include=FALSE` for the first code chunk of an R Markdown document, where we are calling `library`, initializing variables, and doing other housekeeping tasks whose incidental output is just an annoyance.

In addition to the output options just described, there are several options that control handling of the error messages, warning messages, and informational messages generated by your code:

- `error=TRUE` allows your document to build completely even if there is an error in the code chunk. This is helpful when you’re creating a document where you specifically want to see the error in the output. The default is `error=FALSE`.
- `warning=FALSE` suppresses warning messages. The default is `warning=TRUE`.
- `message=FALSE` suppresses informational messages. This is handy when your code uses chatty packages that produce messages while loading. The default is `message=TRUE`.

## See Also

The [R Markdown cheat sheet from RStudio](#) lists many available options.

The author of `knitr`, Yihui Xie, has documented the options on his [website](#).

## 16.8 Inserting a Plot

### Problem

You want to insert a plot into your output document.

### Solution

Simply create a code chunk that creates the plot, and insert that chunk into your R Markdown document. R Markdown will capture the plot and insert it into your output document.

### Discussion

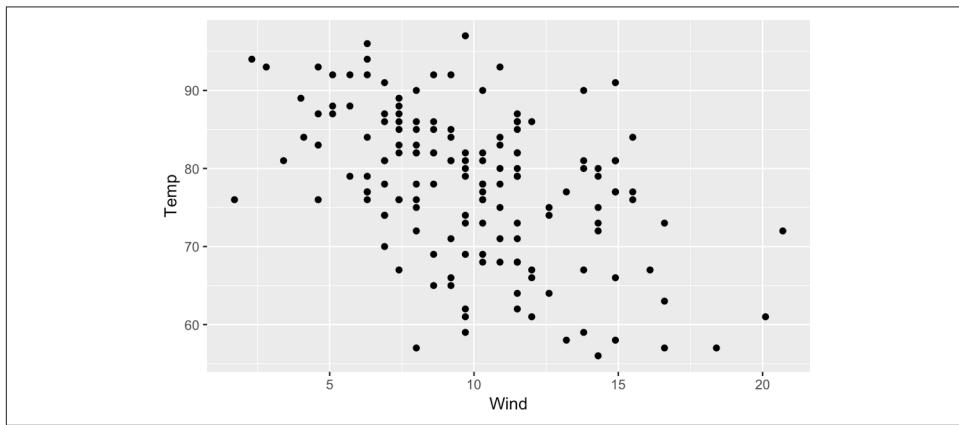
Here’s an R Markdown code chunk that creates a `ggplot` plot called `gg`, then “prints” it:

```
```{r}
library(ggplot2)
gg <- ggplot(airquality, aes(Wind, Temp)) + geom_point()
print(gg)
```
```

Recall that `print(gg)` renders the plot. If we insert this code chunk into an R Markdown document, R Markdown will capture the result and insert it into the output, which looks something like this:

```
library(ggplot2)
gg <- ggplot(airquality, aes(Wind, Temp)) + geom_point()
print(gg)
```

The resulting plot is shown in [Figure 16-5](#).



*Figure 16-5. Example ggplot in R Markdown*

Almost any plot we can produce in R can be rendered into the output document. We have some control over the rendered results using options in the code block, such as setting the size, resolution, and format of the output. Let's look at some examples using the `gg` plot object we just created.

We can shrink the output using `out.width`:

```
```{r out.width='30%'}
print(gg)
```
```

which results in [Figure 16-6](#):

```
print(gg)
```

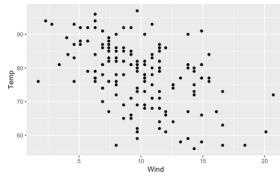


Figure 16-6. Small-width plot

Or we can enlarge the output to the full width of the page:

```
```{r out.width='100%'}
print(gg)
````
```

which results in [Figure 16-7](#):

```
print(gg)
```

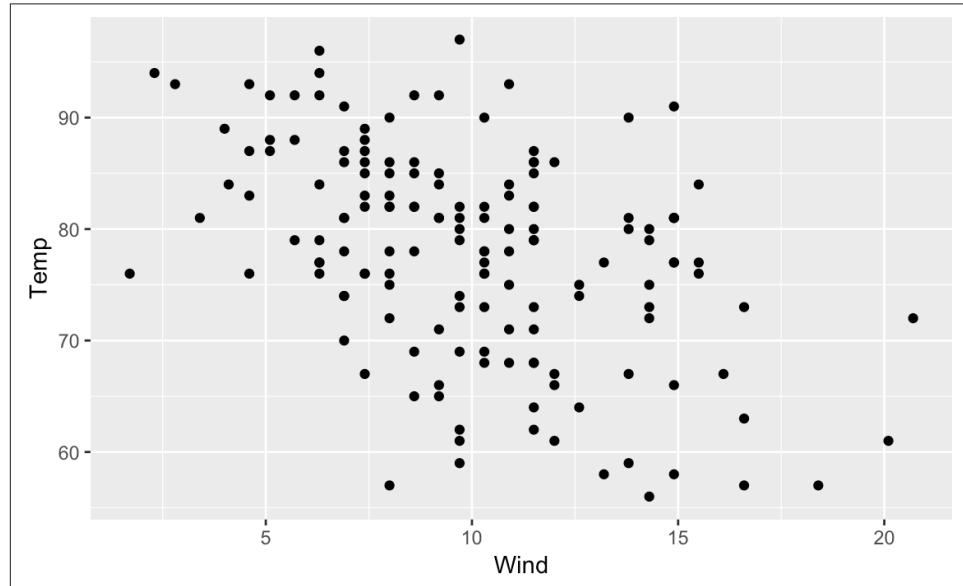


Figure 16-7. Large-width plot

Some common output settings to use with graphics are:

`out.width` and `out.height`

The size of the output figure as a percentage of the page size.

`dev`

The R graphical device used to create the figure. The default is 'png' for HTML output and 'pdf' for LaTeX output. You can also use 'jpg' or 'svg', for example.

`fig.cap`

The figure caption.

`fig.align`

The alignment of the plot: 'left', 'center', or 'right'.

Let's use these settings to create a figure with 50% width, 20% height, a caption, and left alignment:

```
```{r out.width='50%',  
  out.height='20%',  
  fig.cap='Temperature versus wind speed',  
  fig.align='left'}  
print(gg)
```

This produces [Figure 16-8](#):

```
print(gg)
```

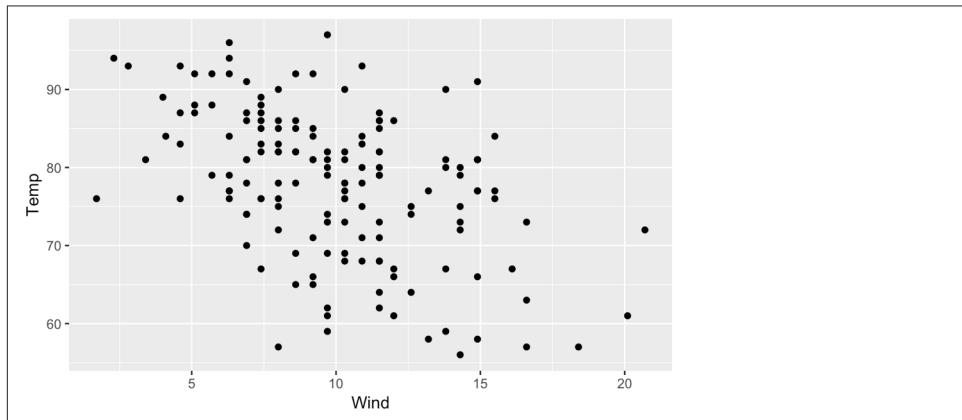


Figure 16-8. Temperature vs. wind speed

16.9 Inserting a Table

Problem

You want to insert a nicely formatted table into your document.

Solution

Lay out the contents in a text table, using the pipe character (|) to separate columns. Use dashes to “underline” column headings. R Markdown will format that into attractive output. For example, this input:

Stooge	Year	Hair?
Moe	1887	Yes
Larry	1902	Yes
Curly	1903	No (ironically)

will produce this output:

Stooge	Year	Hair?
Moe	1887	Yes
Larry	1902	Yes
Curly	1903	No (ironically)

You *must* place a blank line before and after the table.

Discussion

The syntax for tables lets you “draw” the table using ASCII characters. The “underline” made from dashes is a signal to R Markdown that the line above it contains column headings. Without that “underline,” R Markdown would interpret the first line as contents, not headings.

The table formatting is a bit more flexible than the Solution might suggest. This (ugly) input, for example, would produce the same (beautiful) output as shown in the Solution:

Stooge	Year	Hair?
Moe	1887	Yes
Larry	1902	Yes
Curly	1903	No (ironically)

The computer cares only about pipe characters (|) and dashes. The whitespace padding is optional. Use it to make the input easier for you to read.

A handy feature is the use of colons (:) to control justification of columns. Include colons in the dash “underline” to set the column justification. This table defines the justification for three of four columns:

Left	Right	Center	Default
12345	12345	12345	12345
text	text	text	text
12	12	12	12

which gives this result:

Left	Right	Center	Default
12345	12345	12345	12345
text	text	text	text
12	12	12	12

Use the colons within a column heading’s “underline” this way:

- A colon at the extreme left end causes left justification.
- A colon at the extreme right causes right justification.
- Colons at both ends cause center justification.

See Also

Actually, R Markdown supports several syntaxes for tables—some might say a *bewildering* number of syntaxes. This recipe shows only one, just to keep it simple. See the Markdown reference material for the alternatives.

16.10 Inserting a Table of Data

Problem

You want to include a table of computer-generated data in your output document.

Solution

Use the `kable` function from the `knitr` package, shown here formatting a data frame called `dfrm`:

```
library(knitr)
kable(dfrm)
```

Discussion

In Recipe 16.9, we showed how to put a static table into a document using plain text. Here, we have the table contents captured in a data frame, and we want to show the data in the document output.

We could just print the table, and it would end up in the output, unformatted:

```
myTable <- tibble(  
  x=c(1.111, 2.222, 3.333),  
  y=c('one', 'two', 'three'),  
  z=c(pi, 2*pi, 3*pi))  
myTable  
#> # A tibble: 3 x 3  
#>   x     y     z  
#>   <dbl> <chr> <dbl>  
#> 1  1.11 one   3.14  
#> 2  2.22 two   6.28  
#> 3  3.33 three 9.42
```

But we typically want something more attractive and formatted. The easiest way to implement this is by using the `kable` function from the `knitr` package (Figure 16-9):

```
library(knitr)  
kable(myTable, caption = 'My Table')
```

My Table		
x	y	z
1.11	one	3.14
2.22	two	6.28
3.33	three	9.43

Figure 16-9. A `kable` table

The `kable` function takes a data frame as input and a number of formatting parameters, returning a formatted table suitable for display.

`kable` produces great-looking output, but many people discover they want more control over the output than it allows. Luckily `kable` can be paired with another package, `kableExtra`, for—not surprisingly—extra `kable` functionality.

Here we set the rounding and caption using `kable`. Then we use `kable_styling` to make the table more narrow than full width, add shaded striping in our LaTeX output, and center the table in the output (Figure 16-10):

```

library(knitr)
library(kableExtra)
#>
#> Attaching package: 'kableExtra'
#> The following object is masked from 'package:dplyr':
#>
#>     group_rows

kable(myTable, digits = 2, caption = 'My Table') %>%
  kable_styling(full_width = FALSE,
    latex_options = c('hold_position', 'striped'),
    position = "center",
    font_size = 12)

```

x	y	z
1.11	one	3.14
2.22	two	6.28
3.33	three	9.42

Figure 16-10. A `kableExtra` table

The `kable_styling` function takes a `kable` table as input (not a data frame), plus formatting parameters, then returns a formatted table.

Some options in `kable_styling` have a different impact on your output depending on your output format. In our previous example, the `full_width = FALSE` does not change anything in LaTeX (PDF) format because tables in LaTeX output default to not being full width. In HTML, however, the default behavior for `kable` tables is to be full width, so this option has an impact.

Similarly, the `latex_options = c('hold_position', 'striped')` option applies only to LaTeX output, not HTML. The '`hold_position`' ensures that the table ends up where we put it in our source, not at the top or bottom of the page, which tends to happen in LaTeX. The '`striped`' option makes zebra-striped tables with alternating light and dark rows for easier reading.

For more control over Microsoft Word tables, we recommend using the function `flextable::regularTable`, which is discussed in [Recipe 16.14](#).

16.11 Inserting Math Equations

Problem

You want to insert a mathematical equation in your document.

Solution

R Markdown supports the LaTeX math equation notation. There are two ways of entering LaTeX in R Markdown.

For short formulas, put the LaTeX notation inline between single dollar signs (\$). The notation for the solution to a linear regression could be expressed as $\beta = (X^T X)^{-1} X^T \mathbf{y}$, which would result in the inline formula $\beta = (X^T X)^{-1} X^T \mathbf{y}$.

For large formula blocks, embed the block between double dollar signs (\$\$), like this:

```
$$
\frac{\partial C}{\partial t} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 C}{\partial C^2} + rS \frac{\partial C}{\partial S} = rC
\label{eq:1}
$$
```

which generates this output:

$$\frac{\partial C}{\partial t} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 C}{\partial C^2} + rS \frac{\partial C}{\partial S} = rC$$

Discussion

The math equation markup syntax is a LaTeX standard that originated in TeX. Building on that standard, R Markdown can render mathematical expressions in PDF, HTML, MS Word, and MS PowerPoint documents. The PDF and HTML formats support a full range of LaTeX math equations. The translation into Microsoft Word and PowerPoint, however, supports only a subset of the full syntax.

The details of LaTeX equation notation are beyond the scope of this book, but since TeX has been around for 40+ years there are many great resources available online and in print. A very good online resource is the [Wikibooks.org introduction to LaTeX/Mathematics](#).

16.12 Generating HTML Output

Problem

You would like to create a HyperText Markup Language (HTML) document from an R Markdown document.

Solution

In RStudio, click on the down arrow next to the button labeled Knit at the top of the code editing window. When you do, you'll get a drop-down list of all the output formats available for your current document. Select the “Knit to HTML” option, as shown in [Figure 16-11](#).

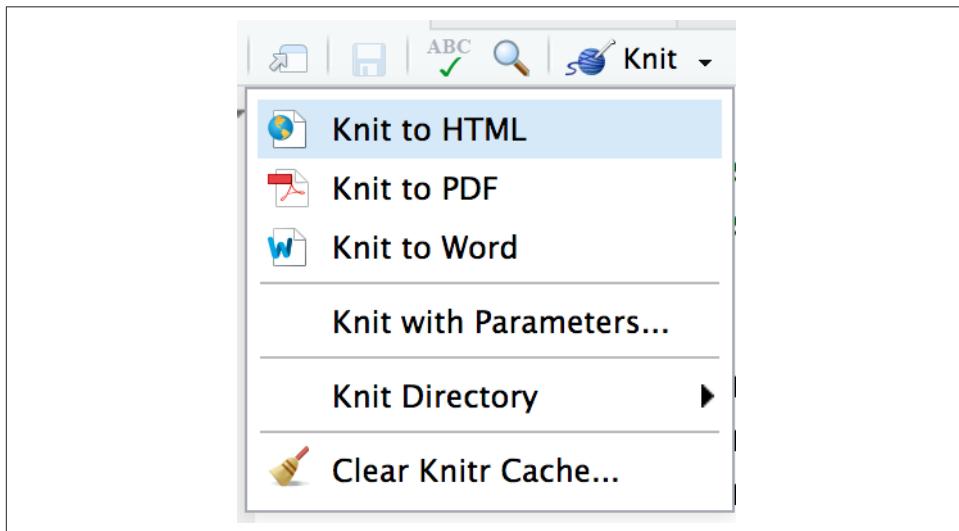


Figure 16-11. Knit to HTML

Discussion

When you select “Knit to HTML,” RStudio moves `html_document: default` to the top of your YAML output chunk at the top of the document, saves the file, and then runs `rmarkdown::render("./YourFile.Rmd")`. If you have knitted your document into three different formats, your YAML may look like this:

```
output:  
  html_document: default  
  pdf_document: default  
  word_document: default
```

If you run `render("./YourFile.Rmd")` on your R Markdown document, substituting your actual filename for `YourFile.Rmd`, it will, by default, knit to the topmost output format (in this case, HTML).



If you are knitting to HTML, your R Markdown document should not contain any special LaTeX-specific formatting, as this will not knit properly in HTML. The exception, as mentioned in prior recipes, is LaTeX math equations, which show up properly in HTML thanks to the MathJax JavaScript library.

See Also

See [Recipe 16.11](#).

16.13 Generating PDF Output

Problem

You would like to create an Adobe Portable Document Format (PDF) document from an R Markdown document.

Solution

In RStudio, click on the down arrow next to the button labeled Knit at the top of the code editing window. When you do, you'll get a drop-down list of all the output formats available for your current document. Select the “Knit to PDF” option, as shown in [Figure 16-12](#).

This will move `pdf_document` to the top of your YAML `output` options:

```
---
```

```
title: "Nice Title"
output:
  pdf_document: default
  html_document: default
---
```

and then knit the document to PDF.

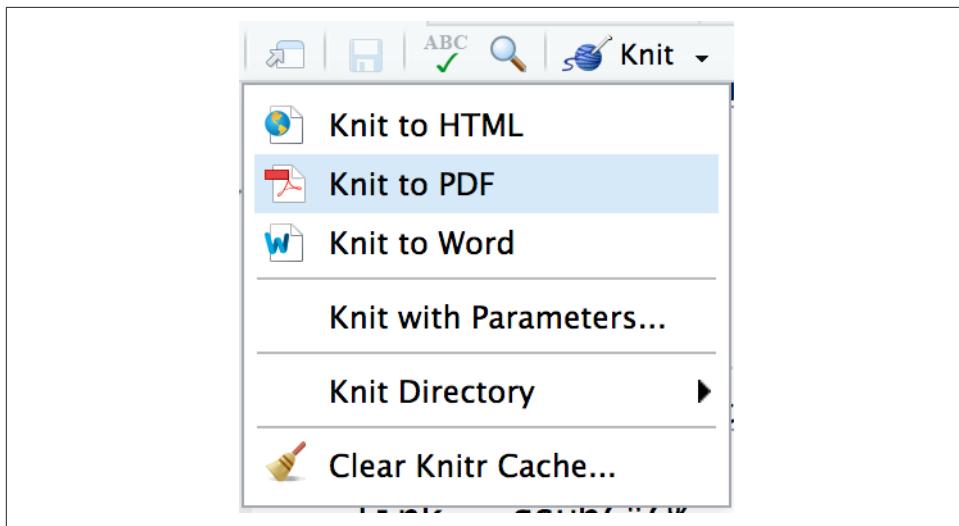


Figure 16-12. Knit to PDF

Discussion

Knitting to PDF uses Pandoc and a LaTeX engine to generate a PDF document. If you don't already have a LaTeX distribution installed on your computer, the easiest way to get one is with the `tinytex` package. Install `tinytex` in R, then call `install_tinytex()`, and `tinytex` will install a small and efficient LaTeX distribution on your computer:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

LaTeX is rich with options, and fortunately, most things that we want to do can be represented with R Markdown and automatically converted to LaTeX via Pandoc. Since LaTeX is a powerful typesetting tool, it is possible to do things with it for which there is no R Markdown equivalent. We can't enumerate all the possibilities here, but we can talk about the ways to pass parameters directly to LaTeX from R Markdown. Keep in mind, though, that any LaTeX-specific options you use will not be translated properly into other formats, like HTML or MS Word.

There are two main ways to pass information from R Markdown to the LaTeX rendering engine:

1. Pass LaTeX directly to the LaTeX compiler.
2. Set LaTeX options in the YAML header.

If you want to pass LaTeX commands directly through to the LaTeX compiler, you can use the LaTeX command beginning with `\`. The limitation is that if you knit the

document to any format other than PDF, the command following the slash is completely omitted from the output.

For example, if we put this phrase into our R Markdown source:

```
Sometimes you want to write directly in \LaTeX !
```

it will be rendered as shown in [Figure 16-13](#).

Sometimes you want to write directly in \LaTeX !

Figure 16-13. \LaTeX typeset

However, if you render your document to HTML, the `\LaTeX` command will be dropped completely, leaving you with an unappealing blank in your document.

If you want to set global options for \LaTeX , you can do so by adding parameters to the YAML header in your R Markdown document. The YAML header has top-level metadata as well as subdata for some options. Different parameters are set at different levels of indentation, so we typically look them up in [*R Markdown: The Definitive Guide*](#) just to be sure.

For example, if you have some previously written \LaTeX content and you want to include it in your document, you can add this prewritten content in three different places in your document: in the header, before the body content, or after the body content at the end. If you were adding external content in all three sections, your YAML header would look something like this:

```
---
title: "My Wonderful Document"
output:
  pdf_document:
    includes:
      in_header: header_stuff.tex
      before_body: body_prefix.tex
      after_body: body_suffix.tex
---
```

Another common \LaTeX option to use is a \LaTeX template for formatting your document. Many templates are available [online](#), and some companies and schools have their own templates. If you want to use an existing template, you can reference it in the YAML header like this:

```
---
title: "Poetry I Love"
output:
  pdf_document:
    template: i_love_template.tex
---
```

You can also turn on or off page numbering and section numbering:

```
---
```

```
title: "Why I Love a Good ToC"
output:
  pdf_document:
    toc: true
    number_sections: true
---
```

Some LaTeX options, however, get set with top-level YAML metadata:

```
---
```

```
title: "Custom Report"
output: pdf_document
fontsize: 12pt
geometry: margin=1.2in
---
```

So when you are setting LaTeX options, consult the R Markdown documentation to determine if the option you are setting is a suboption of the `output:` parameter or its own top-level YAML option.

See Also

See the section “PDF document” in [R Markdown: The Definitive Guide](#).

See also the [Pandoc template documentation](#).

16.14 Generating Microsoft Word Output

Problem

You would like to create a Microsoft Word document from an R Markdown document.

Solution

In RStudio, click on the down arrow next to the button labeled Knit at the top of the code editing window. When you do, you’ll get a drop-down list of all the output formats available for your current document. Select the “Knit to Word” option, as shown in [Figure 16-14](#).

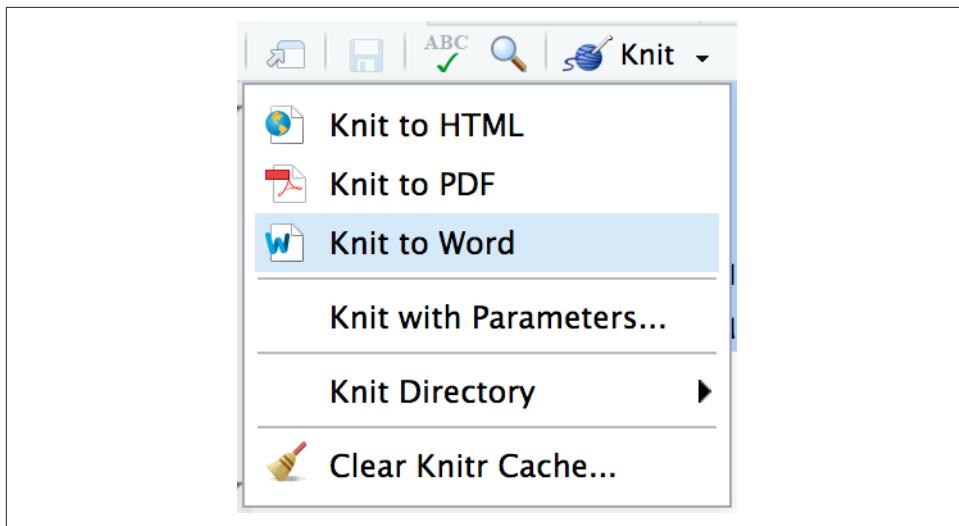


Figure 16-14. Knit to Word

This will move `word_document` to the top of your YAML output options and then knit your R Markdown document to Word:

```
---
```

```
title: "Nice Title"
output:
  word_document: default
  pdf_document: default
---
```

Discussion

Knitting to Microsoft Word is helpful in businesses and scholastic environments where supervisors and collaborators expect documents in Word format. Most R Markdown features work very well in Word, but there are a few tweaks we have found to be helpful when using Word output.

Microsoft has its own equation editing tool. Pandoc will coerce your LaTeX equations into MS Equation Editor, which works well with most basic equations but does not support all LaTeX equation options. One challenge is that MS Equation Editor does not support changing fonts for part of an equation. As a result, matrix notation with fractions and other formulas that require varying fonts can look a bit odd in Word.

Here's a matrix example that looks good in HTML and PDF:

```
$$
M = \begin{bmatrix}
\frac{1}{6} & \frac{1}{6} & 0 & \\[0.3em]
\frac{7}{8} & & \frac{2}{3} & \\[0.3em]
\end{bmatrix}
```

```

          0           & \frac{7}{9} & \frac{7}{7}
\end{bmatrix}
$$

```

Here's how it renders in these output formats:

$$M = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & 0 \\ \frac{7}{8} & 0 & \frac{2}{3} \\ 0 & \frac{7}{9} & \frac{7}{7} \end{bmatrix}$$

But it looks like [Figure 16-15](#) in MS Word.

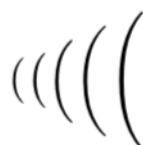
$$M = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & 0 \\ \frac{7}{8} & 0 & \frac{2}{3} \\ 0 & \frac{7}{9} & \frac{7}{7} \end{bmatrix}$$

Figure 16-15. Matrix in MS Word

Any formula using scaling of characters will not work properly in Word. For example, this:

```
$(& \big( \Big( \bigg( \Bigg($
```

would look like this in HTML and LaTeX:



but will get simplified in MS Equation Editor, as shown in [Figure 16-16](#).

(((((

Figure 16-16. Equation font scaling in MS Word

The easiest solution for equations in Word is to try your equation first. If you don't like the output, take your LaTeX equation to an [online free equation editor](#), render it there, and save it as an image file. Then include that image file in your R Markdown document, ensuring that your Word documents have rendered equations that look as good as HTML or LaTeX documents. You will probably want to save your LaTeX equation source in a text file just to make sure you can alter it easily later.

Another challenge with Word output is that often figures don't look quite as good as they do in HTML or PDF. Take this example of a line graph:

```
```{r}
mtcars %>%
 group_by(cyl, gear) %>%
 summarize(mean_hp=mean(hp)) %>%
 ggplot(., aes(x = cyl, y = mean_hp, group = gear)) +
 geom_point() +
 geom_line(aes(linetype = factor(gear))) +
 theme_bw()
```
```

In a Word document this image appears as shown in Figure 16-17.

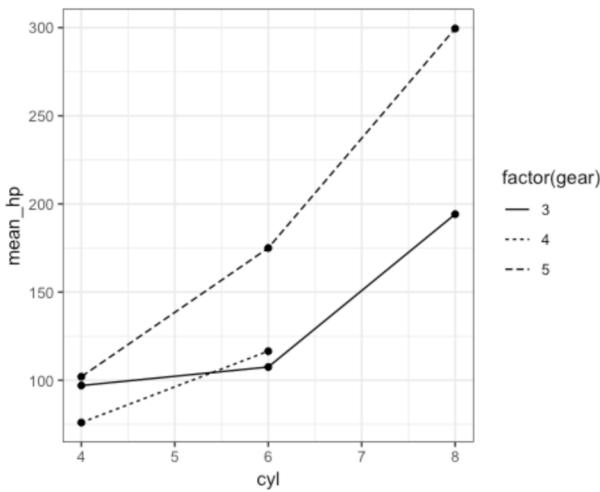


Figure 16-17. Graph in Word

This looks pretty good, but when printed, the image looks a little blocky and not sharp.

You can improve this by increasing the dots per inch (`dpi`) setting used when knitting the output. This will help make the output smoother and sharper:

```
```{r, dpi=300}
mtcars %>%
 group_by(cyl, gear) %>%
 summarize(mean_hp=mean(hp)) %>%
 ggplot(., aes(x = cyl, y = mean_hp, group = gear)) +
 geom_point() +
 geom_line(aes(linetype = factor(gear))) +
 theme_bw()
```
```

To show the improvement in appearance, we've stitched together a composite image showing the default low `dpi` on the left and the higher `dpi` on the right in [Figure 16-18](#).

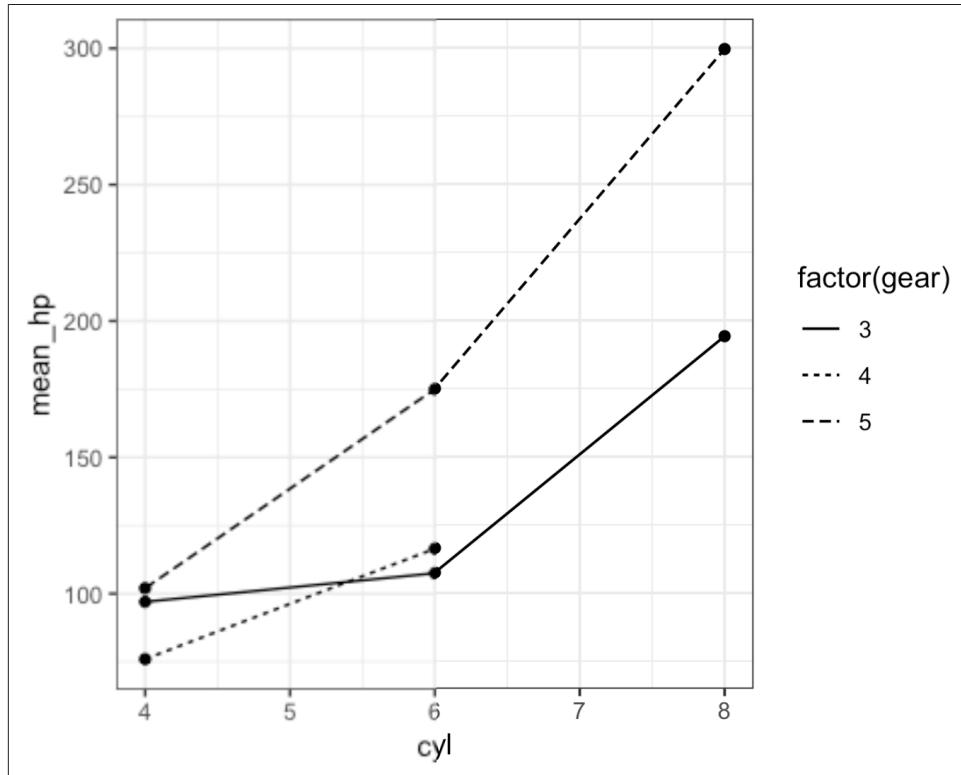


Figure 16-18. Image resolution in Word (default low dpi in left half, higher dpi on right)

In addition to images, table output in Word sometimes is not as customized as we desire. Using `kable`, as illustrated in previous recipes, produces a good, no-frills table in MS Word (see [Figure 16-19](#)):

```
library(knitr)
myTable <- tibble(x = c(1.111, 2.222, 3.333),
                  y = c('one', 'two', 'three'),
                  z = c(5, 6, 7))
kable(myTable, caption = 'My Table in Word')
```

| <i>My Table in Word</i> | | |
|-------------------------|-------|---|
| x | y | z |
| 1.111 | one | 5 |
| 2.222 | two | 6 |
| 3.333 | three | 7 |

Figure 16-19. A table in Word

Pandoc puts the table in a Microsoft table structure inside the Word document. But, just like with tables in PDFs or HTML, we can use the `flextable` package in Word too:

```
library(flextable)
 regulartable(myTable)
```

which gives us [Figure 16-20](#) in Word.

| x | y | z |
|-------|-------|-------|
| 1.111 | one | 5.000 |
| 2.222 | two | 6.000 |
| 3.333 | three | 7.000 |

Figure 16-20. A regulartable in Word

We can tap into the rich formatting features of `flextable` and pipe chains to adjust the column widths, add background color to our headers, and make the header font white:

```
regularTable(myTable) %>%  
  width(width = c(.5, 1.5, 3)) %>%  
  bg(bg = "#000080", part = "header") %>%  
  color(color = "white", part = "header")
```

which gives us [Figure 16-21](#) in Word.

| x | y | z |
|-------|-------|-------|
| 1.111 | one | 5.000 |
| 2.222 | two | 6.000 |
| 3.333 | three | 7.000 |

Figure 16-21. A customized regulartable in Word

For details on all the customizable options in `flextable`, see the `flextable` vignettes and the `flextable` online documentation.

Knitting to Word allows a template to control the formatting of your Word output. To use a template, add `reference_docs: template.docx` to the YAML header:

```
title: "Nice Title"  
output:  
  word_document:  
    reference_docx: template.docx
```

When you knit an R Markdown file to Word using a template, `knitr` maps the formatting of elements in your source document to styles in the template. So if you want to change the font of the body text, you can set the body text style in a Word template to your desired font. Then `knitr` will use the template style in the new document.

A common workflow when using a template for the first time is to knit your document to Word without a template, then open the resulting Word document, adjust the styles of each section to your preference, and use the adjusted Word document as a template in the future. This way, you don't have to guess what style `knitr` is using for each element.

See Also

See the `flextable` vignette on formatting, `vignette('format','flextable')`, and the `flextable` [online documentation](#).

16.15 Generating Presentation Output

Problem

You would like to create a presentation from an R Markdown document.

Solution

R Markdown and `knitr` support creating presentations from R Markdown documents. The most common formats for presentations are HTML (using the `ioslides` or `Slidy` HTML templates), PDF with Beamer, or Microsoft PowerPoint. The biggest difference between R Markdown documents and R Markdown presentations is that presentations default to landscape layout (wide, not long), and every time you create a second-level header starting with `##`, `knitr` will create a new “page” or slide.

The easiest way to get started with presentations with R Markdown is to use RStudio and select File → New File → R Markdown..., then choose one of the four presentation formats offered by the dialog in [Figure 16-22](#).

The four classes of presentations map to the three major classes of documents discussed in previous document recipes.

When it comes time to knit your document to an output format, in RStudio you click the down arrow next to the Knit button and select the type of presentation you would like to produce from the drop-down list, as shown in [Figure 16-23](#).

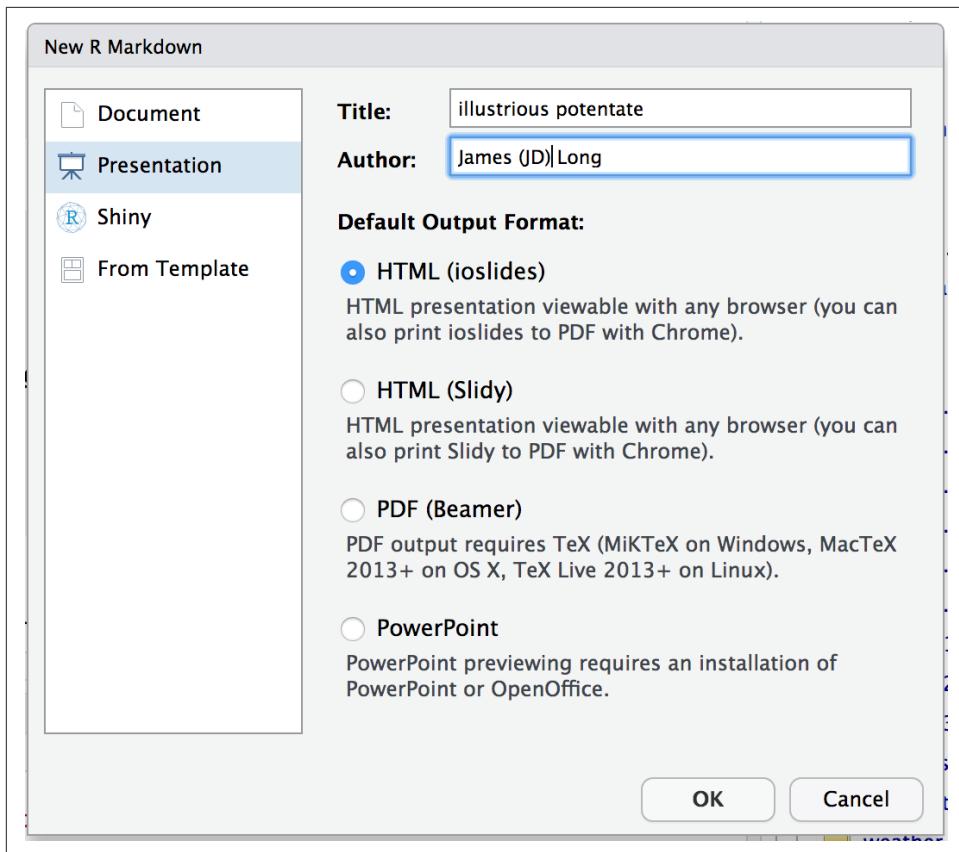


Figure 16-22. New R Markdown Presentation dialog

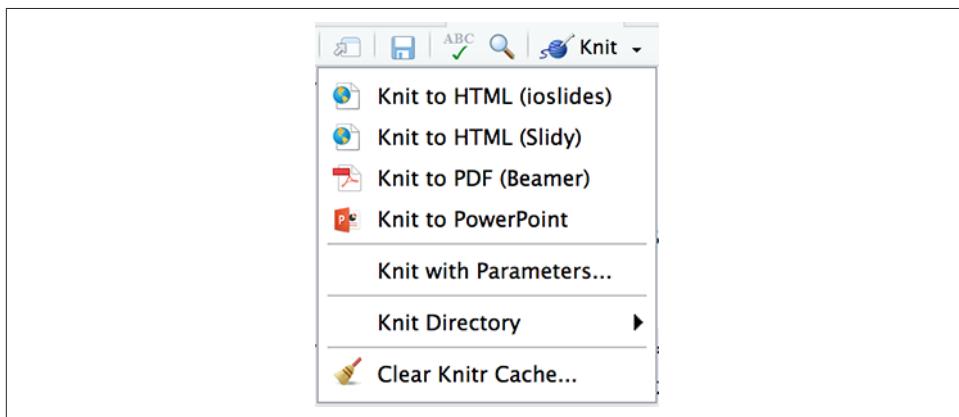


Figure 16-23. Knit: presentations

Discussion

Knitting to a presentation format is very similar to knitting to a regular document, only with different output names. When you use the Knit button in RStudio to choose your output format, RStudio moves your selected output format to the top of the output options in the YAML header of your document, then runs `rmark down::render("your_file.Rmd")`, which knits to the topmost format in your YAML header.

For example, if we selected “Knit to PDF (Beamer)”, the header of the presentation might look like this:

```
---
```

```
title: "Best Presentation Ever"
```

```
output:
```

```
  beamer_presentation: default
```

```
  slidy_presentation: default
```

```
  ioslides_presentation: default
```

```
  powerpoint_presentation: default
```

```
---
```

Most of the HTML options discussed in previous recipes apply to Slidy and ioslides HTML presentations. Beamer is a PDF-based format, so most LaTeX and PDF options discussed in previous recipes apply to Beamer. And last, but never least, PowerPoint is a Microsoft format, so the caveats and options discussed previously about Word documents apply to PowerPoint as well.

See Also

The other recipes related to R Markdown output can be helpful: see [Recipe 16.12](#), [Recipe 16.13](#), and [Recipe 16.14](#).

16.16 Creating a Parameterized Report

Problem

You would like to run the same report periodically with different inputs.

Solution

R Markdown documents can be created with parameters in the YAML header that can then be used as variables in the document body. The parameters are stored as named items in a list called `params`, which you can access in your code chunk:

```
---
```

```
output: html_document
```

```
params:
```

```
  var: 2
```

```
---
```

```
```{r}
print(params$var)
```
```

Later, if you want to change the parameter(s), you have three options:

- Edit the R Markdown document and then render it again.
- Render the document from within R using the command `rmarkdown::render`, passing parameters as a list:

```
rmarkdown::render("test_params.Rmd", params = list(var=3))
```

- Using RStudio, select Knitr → Knit with Parameters, and RStudio will prompt you for parameters before knitting.

Discussion

Using parameters in R Markdown is very helpful if you have a document you need to run regularly with different settings. A common use case is a report in which only a date setting and a label are changed each time it runs.

Here's an example R Markdown document illustrating how parameters can be passed into the text of a document:

```
---
title: "Example of Params"
output: html_document
params:
  effective_date: '2018-07-01'
  quarter_num: 2
---

## Illustrate Params
```{r, results='asis', echo=FALSE}
cat('### Quarter', params$quarter_num,
 'report. Valuation date:',
 params$effective_date)
```

```

The rendered R Markdown results in [Figure 16-24](#).

Example of Params

1 Illustrate Params

1.1 Quarter 2 report. Valuation date: 2018-07-01

Figure 16-24. Parameter output

In the header of the chunk, we set `results='asis'`, because our code chunk is going to generate Markdown text directly. We want to dump that Markdown into our document without prefixing it with `##`, which is what normally happens to the output from a code chunk. In addition, inside the code block we use `cat` to concatenate our text together. We use `cat` here instead of `paste` because `cat` performs less conversion on the text than calling `paste`. This ensures that the text is simply put together and passed into the Markdown document without being altered.

If we want to render the document with other parameters, we can edit the default values in the YAML header and then knit, or we can use the Knitr menu (Figure 16-25) to knit with parameters.

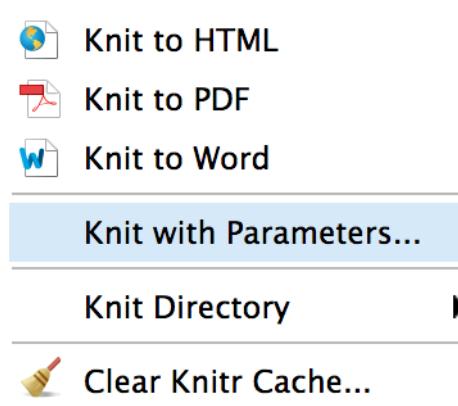


Figure 16-25. Knit with Parameters... menu option

This then prompts us for parameters, as shown in Figure 16-26.

Or we can render the document from R, passing new parameters as a list:

```
rmarkdown::render("example_of_params.Rmd",
  params = list(quarter_num=2, effective_date='2018-07-01'))
```

As an alternative to using the Knitr menu, if we want to be prompted for parameters we can set `params="ask"` when we call `rmarkdown::render` and R will prompt us for inputs:

```
rmarkdown::render("example_of_params.Rmd", params="ask")
```

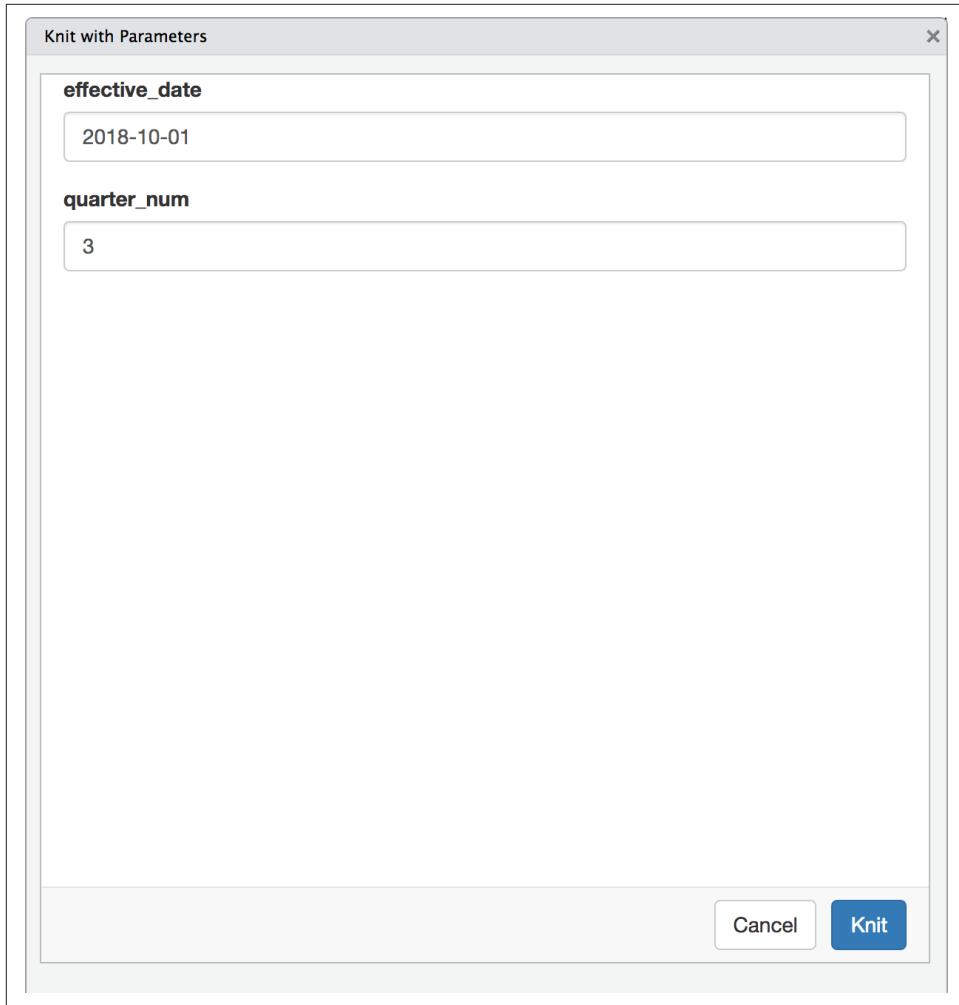


Figure 16-26. Knit with Parameters dialog

See Also

See the section “Parameterized reports” in *R Markdown: The Definitive Guide*.

16.17 Organizing Your R Markdown Workflow

Problem

You want to organize your R Markdown project so that it's efficient, flexible, and productive.

Solution

The best way to get control of your project is to organize your workflow. Organization takes a bit of effort, so it might be overkill to have a highly structured project if your R Markdown document is only one page of output with three small code chunks. However, most people find that organizing their workflow is worth the added effort.

Here are four tips for organizing your workflow so that your work is easier to read, edit, and maintain in the future:

1. Use RStudio Projects.
2. Name directories intuitively.
3. Create an R package for reused logic.
4. Keep R Markdown focused on content, and source logic.

Use RStudio Projects

RStudio includes the notion of an RStudio Project (note the capital P), which is a way of storing metadata and settings related to a logical project. When you open a Project in RStudio, one of the things that RStudio does is set the working directory to the path where the Project is located. Every Project should live in its own unique directory. All code is run from that working directory, which means your code should never contain `setwd` commands that would keep your analysis from being run on someone else's computer.

Name directories intuitively

It's a good idea to organize the files in your Project directory into subdirectories and then name your files thoughtfully inside those directories. As the number of files in a project increases, so does the importance of organization and intuitive naming. One common structure recommended by the team at [Software Carpentry](#) is this:

```
my_project
|- data
|- doc
|- results
|- src
```

In this structure, raw input data goes in the *data* directory, documentation goes in *doc*, results of analyses go in *results*, and R source code goes in *src*.

Once you have a directory structure to put your work into, the individual files should be named in a way that's readable to both humans and computers. This helps with maintaining your code in the future and saves a lot of headaches. Some of the best advice we've seen on file naming comes from [Jenny Bryan](#):

- Use underscores instead of spaces in filenames; spaces cause too many headaches later.
- If you put dates in your filenames, use ISO 8601 dates: YYYY-MM-DD.
- Use a prefix on your scripts so they sort properly—for example, *00_start_here.R*, *01_data_scrub.R*, *02_report_output.Rmd*.

Using numeric prefixes on your scripts and using ISO 8601 dates helps ensure that your files sort in a meaningful way by default. This is very helpful when someone else, or even future you, tries to make sense of your project.

Create an R package for reused logic

Once you have a good directory structure and rational naming, you should give some thought to what logic goes where. You should consider building an R package for logic you use in more than three different projects. R packages are collections of functions and other code that provide functionality not available in Base R. Throughout this book we've used a number of packages, and there's nothing stopping you from writing a package for your functions that you use over and over. Building a package is out of scope for this book, but Jim Hester's presentation "[You Can Make a Package in 20 Minutes](#)" is one of the best introductions to the topic.

Keep R Markdown focused on content, and source logic

Most of us start a project with one big *.Rmd* file full of all our logic in code chunks. As the document grows and the code chunks expand, this can get difficult to manage. You may find that your code formatting is intermingled with code that reshapes data and fetches things from files and databases. Having logic, formatting, and presentation code all intermingled can make it hard to alter your code later and even harder for someone else to understand your code. We recommend keeping the code blocks in your main reporting *.Rmd* file focused on content, tables, and figures and having your manipulation logic stored in **.R* files that you pull in with the `source` function.

Using `source` to pull in external R code involves passing the filename of your R file to the `source` function:

```
source("my_logic_file.R")
```

R will run the entire contents of `my_logic_file.R` at the place in your code where you call `source`. A good pattern is to source files that extract data frames and reshape your data into the form you need to make graphs or tables in your document. Then, in your main `.Rmd` file, you keep mostly code that prepares graphs and tables.

Keep in mind that this is a design pattern for managing large, unwieldy R Markdown files. If your project is not very large, you should probably just keep all your code in the `.Rmd` file.

See Also

Useful references include:

- “Project-oriented workflow” tidyverse documentation
- *Project Management with RStudio* from Software Carpentry
- *R Packages* by Hadley Wickham (O’Reilly)
- *Naming Things* by Jenny Bryan
- “Good Enough Practices in Scientific Computing” by Greg Wilson, et al.

Index

Symbols

- ! (logical negation) operator, 46
- != (inequality) operator, 40, 46
- # beginning comment lines, 100, 102, 416
- #! (shebang) line, 79
- ## ending comment lines, 416
- % (percent sign) in format strings for dates, 206
- %% (modulo) operator, 47, 398
- %*% (matrix multiplication) operator, 47, 155, 413
- %/-% (integer division) operator, 47, 413
- %>% (pipe) operator, 47, 49, 165, 392, 413
- %in% (contained in) operator, 47
- %...% (binary) operator, 47, 413
- & (logical and) operator, 43, 46, 55
- && (short-circuit and) operator, 46, 55
- () (parentheses)
 - enclosing an assignment, 394
 - enclosing conditions in if-else statement, 501
 - enclosing function parameters, 504
 - grouping with, 54, 329
 - in function calls, 52
- * (asterisk)
 - multiplication and inclusion of constituent terms, 351
 - multiplication operator, 44, 46
- + (plus sign)
 - connecting graphical elements in ggplot2, 269
 - connecting graphical elements in patchwork, 328
 - continuation prompt, 8, 53
 - unary plus or addition operator, 44, 46
- , (comma)
 - as decimal mark, 103
 - separator, 101, 103, 193
- (unary minus, subtraction) operator, 42, 44, 46
 - using minus before variable names with select, 172
- text beginning/ending with, in R Markdown documents, 519
- ending comment lines, 416
- > (assignment) operator, 30, 46
- >> (assignment) operator, 46
- . (dot)
 - dot operator, 51
 - indicating missing values, 99
 - names beginning with, 32, 92
 - View tab in RStudio, 392
- / (division) operator, 44, 46
- / (forward slash)
 - in patchwork groupings, 329
 - selecting year and month ranges, 458
- / (forward slash) as path separator, 56, 94
- : (colon) operator, 38, 42, 47, 351
- :: (double colon) operator, 17
- ;(semicolon) separators, 103
- < (less than) operator, 40, 46
- <- (assignment) operator, 29, 46
 - mistakes with, 52
- <-> (assignment) operator, 30, 46
- <= (less than or equal to) operator, 40, 46
- = (assignment) operator, 30, 46
- == (equality) operator, 40, 46
 - mistaking = operator for, 53
- ===== ending comment lines, 416

> (command prompt), 8, 53
> (greater than) operator, 40, 44, 46
>= (greater than or equal to) operator, 40, 46
? (help) operator, 46
?? (search shortcut), 16
[[[]]] (list indexing), 55, 112, 128, 133, 142
accessing single list elements, 144
in list expressions, 166, 169

A

abline function, 320
ACF (see autocorrelation function)
acf function, 473
ADF (Augmented Dickey-Fuller) test, 494
adf.test function, 410, 494
adfTest function, 495
aesthetics, 266
aes function, 267
fill parameter of aes function, 299
grouping parameters in aes function, 282
passing name of categorical variable to aes, 311
shape parameter of aes function, 278
Alt key combinations, 48

finding differences between means of groups, 383-386
on differences between groups, 141
one-way, 334
performing, 380-382
robust, performing (Kruskal-Wallis test), 386
table, 335, 342
anonymous functions, 510
anova function, 334, 388
aov function, 141, 381, 383
append function, 136
inserting data into a vector, 137
Applied Linear Regression Models (Kutner et al.), 335
apply family of functions, 181
apply function, 185, 187
apply.monthly function, 470
args function, 14
arguments, 504
function, taking from a list, 411-413
help with, 14
mistakenly passing multiple arguments to single-argument function, 55
arima function, 483, 485, 486
arithmetic operations
on time series data, 466
on vectors, 44-45
operator precedence, 46
arm package, 369
arrange function, 403
arrays, 131
as.character function, 177, 198, 205
as.complex function, 177
as.data.frame function, 158, 178
using with map, 161
as.Date function, 196, 204, 456
as.integer function, 177
as.list function, 178
as.logical function, 177
as.matrix function, 178, 402
as.numeric function, 177
as.POSIXct function, 196
as.POSIXlt function, 196, 208
as.vector function, 178, 203, 402
as.xts function, 449
as.zoo function, 449
ASCII
dataset stored in, 116

drawing tables using ASCII characters, 530
 saving in ASCII format, 123
 writing tabular data to ASCII file in CSV format, 103
a
 assignment
 -> (assignment) operator, 30
 -<- (assignment) operator, 29
 -<<- (assignment) operator, 30
 = (assignment) operator, 30
 data elements to a vector, 136
 populating list elements via, 148
 printing results of, 394
as_tibble function, 159
 atomic values/data types, 177
attr function, 404
 attributes
 attributes Base R function, 110
 stripping from variables, 404
augment function, 364, 369
 Augmented Dickey-Fuller (ADF) test, 410, 494
 authentication to MySQL databases, 118
 author, setting for R Markdown document, 519
auto.arima function, 482, 485, 486, 488
 autocorrelation
 testing residuals for, 376-378
 testing time series for, 475
 autocorrelation function (ACF), 488, 489
 graphing for residuals, 377
 plotting for a time series, 473-475
autoplot function, 481, 491
 autoregression (AR) coefficients in ARIMA model, 476
 autoregressive integrated moving average (ARIMA) model, 474, 476
 fitting to a time series, 482-486
 making forecasts from, 490
 removing insignificant coefficients, 486
 running diagnostics on, 487-490
 averages, 182, 186
 (see also mean; **mean** function)
 computing moving average of a time series, 467
B
 background grid of ggplot graphics, changing, 270-274
 bandwidth parameter in smoothing data, 496
 bar charts
 adding confidence intervals, 295-298
 coloring or shading, 298-300
 creating, 292-295
 Base R documentation, 12
 batch scripts, running, 77-79
 Beamer (PDF-based format), 548
 Bernoulli trials, generating random sequence of, 222
 beta distributions, 330
 beyond basics (see techniques, advanced)
 binary data
 binary-valued variable, predicting, 436-438
 save function writing, 123
 binary operators
 defining your own, 413
 special meanings inside regression formulas, 359
bind_rows function, 161
 binning data, 396
 binomial coefficients, 216
 binomial distribution, 213
 cumulative probability function **pbinom**, 223
 density function **dbinom**, 223
 pbinom function, 224
 bins (in histograms), 313
 blogdown package, 515
 body of a function, 504
 bookdown package, 515
 Boolean values, 39
 boot function, 439
 boot.ci function, 440
 bootstrap procedures, 247
 bootstrapping a statistic, 438-441
 sampling with replacement, 221
 bootstrap replications, 439
 bootstrap samples, 439
 Box.test function, 475
 boxcox function, 363-368
 boxplots
 creating, 309-311
 creating one for each factor level, 311-313
 of clusters, 436
 Box-Cox procedure, 363-368
 Box-Pierce test for autocorrelation, 475
 broom library, 364, 369
 bulleted lists, 521
C
c operator, 34, 55, 87, 135

calendar period, applying function to time series by, 469-471
call by value, 499
CALL statements (SQL), 119
canonical correlation, 19
car package, 371, 378
Cartesian product, 202, 401-402
case or switch statements, 500
case_when function, 192, 507
cat function, 28
 flattening a list into a vector for, 150
 redirecting output to file with file argument, 90
 redirecting output to file with sink function, 90
 using format function with, 89
categorical variables, 132, 139, 291
 (see also factors)
 in ggplot boxplot, 311
 testing for independence, 239
cbind function, 138, 163, 173, 395, 412
Ccf function, 478
character data, 188, 197
 (see also strings)
 converting atomic values to, 177
 in creation of data frames vs. tibbles, 159, 161
map_chr function, 182
 using character strings for switch labels, 507
checkresiduals function, 488
chi-squared test, 240
chisq.test function, 240
choose function, 215
chooseCRANmirror function, 75
chron package, 196
classes
 defining abstract type of objects, 129
 for dates and time, 195
 deciding which to select, 197
 revealing object's class with class function, 406
cloud, installing R and RStudio in, 84
cluster and multiprocess, remote plan using, 421-423
clusters, finding in data, 433-436
CMD BATCH subcommand, 77
Cmd key combinations, 48
code
 controlling results shown in R Markdown documents, 525
 inserting R code in R Markdown document, 523
 reindenting automatically, 513
 timing running of, 408-410
 using sections, 416
code blocks
 options controlling display in R Markdown, 525
 running in R scripts, 77
 saving in a script, 48
code chunks, 523
coefficients, 333, 407
 (see also regression coefficients)
 autoregression (AR) coefficients in ARIMA model, 476
 calculating coefficient of variation, 504
 for ARIMA model fitted to time series, 483
 for ARIMA model, removing insignificant coefficients, 486
 moving average coefficients, 474
coefplot function, 369
coin toss, 401
 generating random sequence of, 222
collect function, 121
colnames attribute, 156
colon operator (see : (colon) operator, under Symbols)
colors
 adding to ggplot bar chart, 298-300
 color parameter in geometric object functions, 307
 plotting a variable in multiple colors, 322-325
 specifying for lines in a line chart, 302
colSums function, 394
columns
 changing names in data frames, 170
 column as sort key for a data frame, 403
 creating new column in a data frame based on a condition, 192-193
 data in, using to initialize a data frame, 158
 defining width with read_fwf function, 95
 excluding by name in data frames, 172
 in data frames, 133
 in matrices or data frames, applying a function to, 186-188

in tables in R Markdown, setting justification, 531
merging data frames by a common column, 174-176
putting data into and printing, 395
selecting by name in data frames, 168-170
selecting by position in data frames, 165-168
selecting one column from a matrix, 157
combinations
counting number of, 215
generating, 216
generating all combinations of several variables, 401-402
combinations function, 203
combn function, 216
comma-separated values files (see CSV files)
command line
+ prompt, 8
> prompt, 8
editing, 8
starting R from, using --quiet option, 414
commandArgs function, 78
commands
entering, 7-9
shortcuts, 9
saving result of previous command, 65
viewing command history, 64-65
comments, 416
comment parameter in read_table2 function, 100
in read_csv function, 102
compact function, 151
comparison operators ($==$ $!=$ $<$ $>$ \leq \geq), 39, 46
complex atomic type, 177
computer scientists, meaning of data frames to, 134
conditional branch, 500
conditional execution, 500
conditionals
creating new column in a data frame based on a condition, 192-193
if-else statements, 500-502
removing list elements using a condition, 152-153
conditioning plots, 290
conf.level argument, 249
confidence intervals, 235
adding to a bar chart, 295-298
bootstrapping for statistics, 438-441
checking for intercepts in regression, 346
computing limits of, 227
finding for ARIMA model coefficients, 485
for ARIMA model coefficients, 486
for regression coefficients, 341, 368
forming for a mean, 244-245
forming for a median, 246
forming for a proportion, 248
of a correlation, 255
confint function, 368
connection to a file, writing output to, 91
connection to MySQL database, 118
contained in operator (%in%), 47
contingency tables, 238
continuation prompt (+), 53
continuous distributions, 213
calculating probabilities for, 225
quantile functions for, 228
coord_flip function, 310
coplot function, 292
cor function, 348
cor.test function, 255
coredata function, 451, 494
corporate executives, meaning of data frames to, 134
correlation
autocorrelation function of a time series, 473
calculating correlation matrix from a data frame, 172
calculating with cor function, 36
finding lagged correlations between time series, 478
partial autocorrelation function for time series, 476-478
testing for autocorrelation in time series, 475
testing for significance, 255-257
covariance, 339, 375
calculating with cov function, 36
CRAN (Comprehensive R Archive Network)
CRAN Search, 22
crantastic.org, searching for packages by keyword, 21
distributions in downloadable packages, 215
downloading and installing R from, 2-4
installing packages from, 72-73
list of task views, 21
package documentation, 1

packages for dates and time, 196
setting or changing default mirror, 74
cross-correlation function, 478
cross-sectional data, 448
cross-tabulations (see contingency tables)
CSV (comma-separated values) files
 read.csv function, 509
 reading from, 101-103
 reading from the web, 104
 writing to, 103
Ctrl key combinations, 48
cubic spline, 462
cumsum function, 172
cumulative probability function, 224
customizing R startup, 81-84
cut function, 396
cutree function, 433

D

damped sine wave, 327
data
 binning, 396
 computer-oriented representations of, 113
 displaying partial, 391
 entering from the keyboard, 87
 finding clusters in, 433-436
 self-describing, 100
data frames
 accessing content of, 133
 appending rows to, 162-165
 applying a function to each column, 37, 186-188
 applying a function to every row, 184
 changing names of columns, 170
 combining two, 173
 comparison to tibbles, 134
 converting between other structured types, 178
 converting list of vectors to, 141
 converting to z-scores, 243
 creating new column based on a condition, 192-193
 creating vs. creating tibbles, 159
 defined, 133
 different meanings coming from different backgrounds, 133
 excluding columns by name, 172
 flattening, 402
 formatting in R Markdown document, 531

functions not understanding, misusing, 56
in linear regressions, 336
initializing from column data, 158
initializing from row data, 160-162
merging by common column, 174-176
multiple, adding to one plot, 305
removing NA value from, 171
selecting columns by name, 168-170
 using list expressions, 169
 using matrix-style subscripting, 169
 using select function, 169
selecting columns by position, 165-168
 using list expressions, 166
 using matrix-style subscripting, 167
 using select function, 165
sorting, 403
summary of, 236
summing rows and columns, 394
writing to Excel file, 107, 109

data function, 70
data structures, 127
 (see also data frames; factors; lists; matrices; tibbles; vectors)
 representing time series data, 449

data transformations (see transformations)

data types
 dynamic, 30
 in vectors, 34
 R as typeless language, 499
 structured, converting to another type, 178-180
 value having atomic type, converting to another, 177

data.frame command, 115, 158
 converting all factors to characters, 161
 stringsAsFactors parameter, 161

datapasta package, 88

datasets
 built-in, viewing, 70
 calculating quantiles and quartiles of, 240
 example, supplied with R, 24
 factor analysis on, 441-445

Date class, 195

dates and time, 195-211
 classes for, 195
 deciding which to select, 197
 converting a date into a string, 205
 converting a string to a date, 204

converting year, month, and day into a date, 206
creating a sequence of dates, 210
Date object and Date class, 129
dates or datetime in time series analysis, 448
extracting parts of a date, 208
getting the current date, 204
getting the Julian date, 208
indexing time series by date, 456
output of timing code execution, 409
setting date for R Markdown document, 519
time representation in time series analysis, 451
vector of, in time series data representation, 449

dbDisconnect function, 119
dbGetQuery function, 119
DBI backend packages, 122
DBI::dbConnect function, 118
dbinom function, 223
dbplyr package, accessing a database with, 120-122
defensive programming, 509
degrees of freedom (DOF), 320
delimiters, 200
 delimiter argument of strsplit, 201
dendrograms, 433
density estimate, adding to histogram, 315-316
density function, 224
 plotting for probability distributions, 228-231
dependent variables, 336
 (see also response)
detach function, 70
detecting errors, 509
devtools package, 74
dice, rolling, finding all combinations of results, 401
diff function, 225, 465-466, 470
 computing difference of logarithms of prices, 467
digits, formatting for printing, 88-90
dimensions
 assigning to a vector, 155
 dropping or retaining in selecting row or column from a matrix, 157
 giving to a list, 131
 giving to a vector, 130
 of matrix variable, stored in attribute, 404

discard function, 152
discrete distributions, 213
 calculating probabilities for, 223-225
 density functions and distribution functions for, 224
 quantile functions for, 227
display options for R code in R Markdown, 525
dist function, 433, 436
distribution functions, 224, 225
distributions
 getting help on, 215
 names of, 213
 random number generators for, 217
 testing two samples for same distribution, 260

do.call function, 412
Docker, 85
documentation, 1
 installing locally, 4
 searching supplied documentation, 16-17
 supplied with R, viewing, 12-14
Documents directory (Windows), profile scripts in, 81
dpill function, 495
dplyr package, 71, 120
 arrange function, 403
 data transformations with, 181
 first and last functions, 456
 joins, 176
 rename function, 170
dplyr::count function, 67
dplyr::lag function, 464
dput function, 24, 123
dump function, 123
durbinWatsonTest function, 378
Durbin-Watson test, 376-378
dwtest function, 377
 alternative option, 378
dynamically typed languages, 30

E

echo option for code display, 525
eigen function, 428
eigenvalues or eigenvectors, calculating, 428
environment variables
 getting and setting, 415
 setting system environment variables, 81
 system environment variable R_HOME, 80
environments

in R search path for functions, 67
RStudio Environment tab, 63
equality operator (see `==`, under Symbols)
equation editor (Microsoft), 540
error messages, suppressing, 410
error terms (in linear regression), 336
errors
 in R code chunks in R Markdown, 526
 protecting against, 509
 signaling in code, 508
escaping special characters with backslash, 94
`eval` option, 525
example function, 15
Excel
 meaning of data frames to a user, 134
 nested IF statements, 192
 reading data from files, 105-107
 writing R data frames to, 107-109
executing R in parallel
 locally, 417-419
 remotely, 420-423
`expand.grid` function, 203, 401
exponential distributions, 321
expressions
 incorrectly continuing across lines, 53
 using in regression formula, 358-360
`extract2` function, 111

F

`F` statistic, 345, 377
facets
 `facet_wrap` function, 290
 functions for, 266
`factanal` function, 441
factor analysis, performing, 441-445
`factor` function, 139
 levels argument, 140
factors, 132
 constructing data frames from, 158
 conversion of character data to in data
 frame creation, 159, 161
 creating, 139
 creating parallel factor from combined vectors, 140
 summary of, 236
 tabulating and creating contingency tables, 238
 use in building data frames, 133
`FALSE` and `TRUE` logical values, 39
faraway package, 382
`fct_order` function, 294, 298
files
 dealing with Cannot Open File in Windows, 94
 listing, 91, 93
 with complex or irregular structure, reading data from, 113-118
fill in ggplot charts, 299
filling or padding time series data, 460-463
`filter` function, 50
first and last functions, 455
`fitdistr` function, 320
fitting ARIMA model to time series, 482-486
fixed-width records, reading, 94-97
`flexdashboard` package, 515
`flextable` package, 544
floating-point numbers
 `map_dbl` function, 182
 R formatting for output, 88
for loops, 500
 iterating over vector or list elements, 502-503
`forcats` package, 294
`forecast` function, 490
`forecast` package, 478, 482
Forecasting: Principles and Practice (Hyndman and Athanasopoulos), 486
forecasts
 making from an ARIMA model, 490
 plotting a time series forecast, 491
`format` function, 89, 205
format strings for dates, 204
 converting American-style date to ISO standard, 205
formatting document text in R Markdown, 520
`fpp2` package
 `euretail` dataset, 486
 Time-Series object types, 494
`full_join` function, 175, 175
function definition, using `inline`, 511
function keyword, 503
functions, 500
 accessing in a package, 69, 70
 applying rolling function to time series, 471-472
 applying to time series by calendar period, 469-471
 arguments, taking from a list, 411-413

assignment statements using <- operator, 30
computing logarithm or square root of time series, 467
creating anonymous function, 510
creating pipeline of function calls, 49-51
defining, 503-505
defining defaults for parameters, 507
distribution-related, parameter requirements, 214
extracting regression statistics from linear models, 339
finding relevant functions, 21
finding via the search path, 67
getting help on, 14-15
graphing value of, 325-328
local variables in, 505
multiparameter, minimizing or maximizing, 426-428
not understanding data frames, 56
pure, 181
reusable, creating collection of, 511
searching for help in supplied documentation, 16-17
single-argument vs. multiple arguments, 55
single-parameter, minimizing or maximizing, 425
fUnitRoots package, 495
furrr package, 420
future_map function, 419, 420

G

gamma distribution, 316
 dgamma density function, 229
gather function, 303
gcd function, 189, 504
geometric object functions, 265
 geom_abline function, 287
 geom_bar function, 292, 297
 geom_boxplot function, 309
 geom_density function, 315
 geom_histogram function, 313
 geom_hline function, 308
 geom_line function, 301, 305
 linetype, col, and size parameters, 302
 geom_point function, 267, 285, 301, 322
 geom_qq function, 320, 321
 geom_ribbon function, 230
 geom_segment function, 325

geom_smooth function, 284, 286
geom_vline function, 307
getTables function, 106
getwd command, 59
GGally package, 289
ggpairs function, 289
ggplot function, 263
 creating smoothed time series data plot, 496
 piping function call results into, 50
 plot in R Markdown, 526
 plotting regression residuals, 364, 369
ggplot2 package, 71, 263
 defining components of graphs, 265
 long vs. wide data, 266
 notes on the basics, 265
 plotting density function for probability distributions, 228-231
 use to plot time series forecast, 491
ggsave function, 331
Git, using, information on, 84
GitHub
 installing packages from, 73
 patchwork package, 328
 R Markdown source for this book, 515
 R Markdown: The Definitive Guide, 515
 using, information on, 84
glm function, 437
Global Options in RStudio, 10
global variables, 30, 500
 local variables and, 505
Google Cloud Platform, 85
graphics, 263-267
 (see also plots)
 common output settings for in R Markdown, 528
 ggplot2 basics, 265
 in packages other than ggplot2, 266
graphics package, 263
grid package, 331
grid size, 496
grouping
 applying a function to groups of data, 191
 patchwork support for, 329
 using categorical variables, 132
groups
 creating scatter plot of multiple groups, 278-280
 finding differences between means of, 383-386

testing for equal proportions, 257
group_by function, 191
gsub function, 201
gtools package, 203

H

haven library, read_sas function, 110
hclust function, 433
head function, 50, 71, 391, 454
header lines
 column names in, 100
 in CSV files, 101
headings, inserting in R Markdown documents, 521
help
 ? operator, 46
 for functions, 14-15
 for options, 82
 for probability distributions, 215
 for regular expressions, 201
 help function, 17, 71
 searching R mailing lists, 22
 searching web for help on R, 18-21
 sources of, 1
help.search function, 16
help.start function, 12
hidden names, 32
 list.files ignoring, 92
hierarchical cluster dendrogram, 434
hist function, 315
histograms
 adding density estimate to, 315-316
 creating, 313-315
history function, 64
home directory
 locating for R, 79
 profile scripts in, 81
“How to Ask Questions the Smart Way” (Raymond and Moen), 26
HTML
 generating HTML document from R Markdown document, 535
 matrix notation rendered in, 540
 R Markdown output to, 519
 reading data from HTML files, 111-113
 Slidy and ioslides presentations, 548
html_table function, 111-113
hypothesis testing, 233

I

I(...) operator, 358
 surrounding expressions inside regression formula, 359
if-else statements, 500-502
ifelse function, 502
images
 image resolution in Word, 543
 saving plots as image in RStudio, 332
increments
 in seq function for Date objects, 210, 457
 in sequences, 38
indenting code, 513
independent variables, 336
 (see also predictors)
indexing
 creating logical indexing vector to sample every nth element, 398
indexes for xts or zoo objects, 450
lists, 55, 112, 142
 double vs. single brackets, 128, 145
scalars, 130
to select row or column from a matrix, 157
vectors, 41-44
 xts or zoo objects, 456
influence.measures function, 375
influential observations, 375-376
inner_join function, 174
install.packages function, 72, 111
installed.packages function, 68
installing R, 2-4
 in the cloud, 84
installing RStudio, 4
install_github function, 74
integers
 converting atomic values to, 177
 map_int function, 182
 switch labels, 506
integrated development environments (IDEs)
 RStudio, 4
interaction plot, creating, 382-383
interaction terms, 359, 361
 generating with stepwise regression, 356
 performing linear regression with, 350-352
interaction.plot function, 382
intercepts, 336
 calculating in orthogonal regression, 432
 performing linear regression without intercept, 345-346

interquartile range (IQR), 310
interrupting R, 11
intersection of all dates, 460, 462
interval probability, 224, 226
is.character predicate, 152
ISODate function, 206
ISOdatetime function, 207

J

joins
 full_join function, 175
 inner_join function, 174
 left_join function, 175
 right_join function, 175
Julian date, getting, 208

K

k-nearest neighbors algorithm, 508
kable function, 531
kable_styling function, 532
keep function, 153
kernel density estimation (KDE), 316
KernSmooth package, 495
keyboard shortcuts, 48
keyboard, entering data from, 87
keywords (search engine)
 broadening your search with, 17
 documentation for, 13
knitr package, 545
knitting R Markdown documents, 515
Kolmogorov-Smirnov test, 260
kruskal.test function, 334, 386
Kruskal-Wallis test, 380, 386
ks.test function, 260

L

labels
 adding to ggplot graphics, 268
 from SAS file read into data frame, 110
 in switch function, 506
 labs function in ggplot, 327
 parameter in cut function, 397
lag function, 463
lagging a time series, 463, 465, 495
 finding lagged correlations between time
 series, 478
lambda argument, 366
lapply function, 184

last function, 455
.Last.value variable, 65
LaTeX
 Beamer and, 548
 excluding from HTML output, 536
 installing distribution on your computer,
 537
Pandoc coercing equations into MS Equation Editor, 540
passing information from R Markdown to
 LaTeX rendering engine, 537
R Markdown document with data table, 532
using templates to format documents, 538
ways to enter LaTeX in R Markdown, 534
lattice package, 267, 331
 histogram function, 315
layers (in ggplot graphics), 266
lazy evaluation, 500
leading data, 464
left_join function, 175
legends, adding or removing in ggplot graphics,
 280-283
length function, 197
levels (of factors), 132, 139
library function, 25, 54, 67, 69, 71, 82, 196
 using with no arguments to display installed
 packages, 68
line charts
 changing type, width, and color of lines,
 302-304
 plotting a line from x and y points, 300-301
linear models, 153
 full model, 353
 reduced model, 353
Linear Models with R (Faraway), 335
linear regression, 333-380
 and ANOVA, 334
 comparing models, using ANOVA, 388-389
 creating interaction plot, using ANOVA,
 382-383
 diagnosing model fit, 371-373
 finding best power transformation (BoxCox
 procedure), 363-368
 finding differences between means of
 groups, using ANOVA, 383-386
 forming confidence intervals for regression
 coefficients, 368
 forming prediction intervals, 379
 getting regression statistics, 339-342

identifying influential observations, 375-376
multiple, performing, 337-338
performing one-way ANOVA, 380-382
performing robust ANOVA (Kruskal-Wallis test), 386
performing with interaction terms, 350-352
performing without an intercept, 345-346
plotting regression line of a scatter plot, 284-288
plotting regression residuals, 369
predicting new values from the model, 378
regressing on a polynomial, 360
regressing on subset of your data, 357
regressing on transformed data, 361-363
regressing only variables highly correlating to your dependent variable, 347-350
selecting best regression variables, 352-357
simple, performing, 335
testing residuals for autocorrelation, 376-378
understanding the regression summary, 342-345
using expression inside a regression formula, 358-360

lines
newline (\n) character, 28
vertical or horizontal, adding to ggplot graph, 307-308

Linux
/ (forward slash) path separator, 56
Alt and Ctrl key combinations, 48
exiting RStudio, 9
installing R, 3
shebang line starting with #!, 79
starting RStudio, 5
Sys.getenv function results, 80

list expressions
using to select columns by name in data frames, 169
using to select columns by position in data frames, 166

list function, 142
building name/value association list, 147-149

list.files function, 91-93

lists, 128
applying a function to each element, 182-184

applying a function to parallel vectors or lists, 188-191
building name/value association list, 147-149
bulleted or numbered, inserting in R Markdown documents, 521-523
containing vectors or factors, initializing into a data frame, 158
converting between other structured types, 178
creating a matrix from, 131, 132
creating and populating, 142-143
data frames as, 133
elements having different modes, 129
elements of different modes (types), 142
flattening into a vector, 150
indexing, 55, 112
printing with print function, 28
removing an element, 149
removing elements using a condition, 152-153
removing NULL elements from, 151
selecting elements by names, 145
selecting elements by position, 144-145
summary of, 236
tags for list elements, 142
taking function arguments from, 411-413
using list functions and operators to examine an object, 407

Ljung-Box test, 476, 489

lm function, 153, 333, 336, 480
data parameter, 337, 338
examining return value for object structure, 406
for multiple linear regressions, 337
subset parameter, 357
with no intercept, 346

lmtest package, 377

load function, 123

loading, 442

loadWorkbook function, 107

local polynomials, 496

local variables, 499
creating, 505

lopcpoly function, 495, 496

log function, 45, 467

logarithmic transformations, 318, 362

logical expressions, 238

logical operators, 43

mistakes in using, 55
logical values, 39, 178
converting other types to, 177
creating logical indexing vector for sampling every nth element, 398
logistic regression, 436-438
long vs. wide data (*ggplot*), 266, 302
loops, 500
iterating over vector or list elements, 502-503
lower.tri function, 203
ls function, 31
forcing to list hidden names, 32
using with *rm* function, 32
ls.str function, 31
lubridate package, 196

M

Mac systems
R search path vs. Unix search path, 67
shebang line starting with `#!/`, 79
Sys.getenv function results, 80
macOS
/ (forward slash) path separator, 56
Cmd and Opt key combinations, 48
exiting RStudio, 10
installing R, 3
starting RStudio, 5
magrittr package, 111, 413
pipes, 342
mailing lists for R, 2
searching for answer to a question, 22
submitting questions to, 23-26
map function, 182
loops and, 503
using with *as.data.frame*, 161
map2 function, 189, 190
map2_chr function, 190
map2_dbl function, 190
map_chr function, 182, 183
map_dbl function, 37, 182, 183, 347
map_df function, 187
map_int function, 182
Markdown, 520
(see also R Markdown)
MASS package, 215
datasets in, 71
MASS::fitdistr function, 320
match function, 398

math equations, inserting into R Markdown documents, 534
Mathematical Statistics with Applications (Wackerly et al.), 235
matrices, 130
%*% multiplication operator, 47
applying a function to every column, 186-188
applying a function to every row, 185-186
calculating correlation matrix from a data frame, 172
calculating eigenvalues or eigenvectors, 428
converting between other structured types, 178
special considerations, 180
converting data frame into, 402
converting to vector, 402
converting to z-scores, 243
correlation and covariance, 37
flattening into a vector, 203
giving descriptive names to rows and columns, 156
heterogeneous, creating from heterogeneous list, 132
initializing, 154
matrix-style subscripting, 167, 169
performing matrix operations, 155
printing with *print* function, 28
selecting one row or column from, 157
summary of, 236
summing rows and columns, 394
use in building data frames, 133
matrix function, 154
max function, 56
mean
calculating, 35-37
calculating for clusters, 434
calculating in *ggplot*, 293
comparing for two samples, 252-253
confidence interval for, 235
finding differences between means of groups, 383-386
forming confidence interval for, 244-245
in example data for linear regression, 335
in *ggplot* graph line, 308
mean function, 45, 55
pairwise comparisons between group means, 258-259
rolling mean, 468

testing mean of a sample, 243
mean function, 14, 238, 242
 applying to each column of a matrix, 187
 collapsing data frames to matrices for, 402
mean reversion, testing time series for, 492-495
mean_se function, 296
median
 calculating, 35
 calculating for samples, 221
 forming confidence interval for, 246
merge function, 459
messages, suppressing, 410-411, 526
metadata about R Markdown document, 519
Microsoft Word
 generating Word document from R Markdown document, 539-545
 equations in Word, 540
 graphics and charts in Word, 542
 tables in Word, 544
R Markdown output to, 519
 tables in R Markdown output to, 533
Min and Max regression residuals, 343
min function, 56
minimum or maximum
 finding, 399
 maximizing multiparameter functions, 426
 maximizing single-parameter functions, 425
 minimizing multiparameter functions, 426
 minimizing single-parameter functions, 425
 using min and max functions for index values, 461
missing values, 99
 (see also NA values)
mistakes (common), avoiding, 52-56
mm/dd/yyyy date format, 205
mode function, 129, 406
models
 comparing linear models using ANOVA, 388-389
 evaluating with ANOVA, 334
 extracting information from linear models
 using functions, 340
 linear models, 333
 model object returned by lm function, 340
modes, 128
 different modes in list elements, 142
 of vector elements, 35, 127
mondate package, 196
moving average (MA) coefficients, 474
MS Equation Editor, 540
multiprocess and cluster, remote plan using, 421-423
mutate function, 184, 193
mysql client program, 118
MySQL databases
 reading from, 118-120
 RMySQL DBI backend package, 122

N

n:m expressions, 38
NA (not available) values, 36
 filling in, html_table function, 112
 in read_table2 function, 99
 in vectors, 43
 removing from data frame, 171
 removing from list by defining a predicate for discard, 153
 replacing in time series data, 461
na.approx function, 462
na.locf function, 461
na.omit function, 171
 removal of entire rows, 172
na.pad setting, 464
na.spline function, 462
Nabble, 22
name/value association lists, 147-149
names
 changing for columns in data frames, 170
 excluding columns by, in data frames, 172
 for list elements, 128, 143
 getting with names function, 349
 given by lm function to regression coefficients, 405
 giving to matrix rows and columns, 156
 indexing vectors by, 43
 selecting data frame columns by, 168-170
 selecting list elements by, 146
nchar function, 197
negative indexes, 42
nonparametric statistics, 254, 380
normal distributions, 221, 335
 and prediction intervals' response to non-normal distributions, 380
 creating normal Q-Q plot, 317-319
 distribution function pnorm, 225
 example random data using rnorm function, 337
functions for, 213

plotting the density function, 228
quantile function qnorm, 227
random number generator, 218
testing data sample for normality, 249
nortest package, 250
not available values (see NA values)
null hypothesis, 233
NULL values
 assigning to attributes property of variables, 404
 assigning to list elements, 149
 in vectors, 43
 removing from list by defining a predicate for discard, 153
 removing NULL elements from a list, 151
number of trials, 257
numbered lists, 521
numeric data
 converting atomic values to, 177
 generating combinations of numbers, 216

O

object orientation, 500
objects
 choice of object class, 447
 class defining abstract type, 129
 mode, 128
 revealing structure of, 405-408
 saving and transporting R objects, 122-125
 stripping attributes from, 405
observations, 160
 influential, identifying in linear model, 375-376
oneway.test function, 334, 380-382
openxlsx package, 105
operator precedence, 46-47
operators
 having special meaning in regression formulas, 359
 user-defined, 414
Opt key combinations, 48
optim function, 426-428
optimize function, 425
options function, 75, 82
 digits parameter, changing default for, 89
ordinary least-squares (OLS) algorithm, 336
orthogonal regression, 430-433
outer function, 202
outlierTest function, 371

output
 CMD BATCH subcommand, to file, 77
 redirecting to file, 90
 Rscript output to stdout, 78
output format for R Markdown documents, 519

P

p-values, 233, 240, 243, 377, 380
 in linear model, 344
pacf function, 476
packages
 accessing functions in, 69-70
 creating your own package of R functions, 512
datasets in, 71
documentation, 1, 13
for dates and time, on CRAN, 196
from CRAN, installing, 72-73
from GitHub, installing, 73
full name of functions in, 67
getting help on, using help function, 17
installed, not loading with library or require, 54
installed, viewing list of, 68
loaded, displaying via search path, 66-67
loading in profile scripts, 82
 relevant, finding, 21
paired data
 pairwise combinations of strings, 202
 plotting pairs of variables, 288-290
paired observations, 252, 254
 creating scatter plot for, 267-268
pairwise comparisons
 between group means, 258-259
 parallel minimum and parallel maximum for vector elements, 400
pairwise.t.test function, 258
Pandoc, 519, 537
Pandoc Markdown Guide, 523
parallel maximum (pmax function), 399
parallel minimum (pmin function), 399
parallelization
 executing R in parallel locally, 417-419
 executing R in parallel remotely, 420-423
parameterized report, creating in R Markdown, 548-552
parameters
 in function definitions, 504
 setting default values for, 507

partial autocorrelation function (PACF), plotting for time series, 476-478
paste function, 198
 using with outer function, 202
patchwork package, 328
PATH environment variable, 67
patterns, 16
 (see also regular expressions)
 using in searches of supplied documentation, 16
pbisnom function, 224
PDF
 generating PDF document from R Markdown document, 536
 Knit to PDF (Beamer), 548
 matrix notation rendered in, 540
 R Markdown output to, 519
Pearson correlation, 256
permutations
 permutation function, 203
 random permutation of a vector, 222
pexp function, 226
pi, 130
 simulating, 418, 421
pipe operator (%>%), 49
pipes
 creating pipeline of function calls, 49-51
 piping object to RStudio viewer, 392
 using magrittr pipes for linear model, 342
plain-text files, R Markdown documents, 518
plan function, 422
plot function, 290, 370, 385, 452
plots
 adding confidence intervals to a bar chart, 295-298
 adding or removing a grid, 270-274
 adding or removing legends in ggplot, 280-283
 adding title and labels, 268
 adding vertical or horizontal lines, 307-308
 applying themes in ggplot, 274-278
 boxplot of clusters, 436
 changing type, width, and color of lines, 302-304
 coloring or shading a bar chart, 298-300
 creating a bar chart, 292-295
 creating a boxplot, 309-311
 creating a scatter plot, 267-268
 creating interaction plot, 382-383
creating normal quantile-quantile (Q-Q) plot, 317-319
creating one boxplot for each factor level, 311-313
creating one scatter plot for each group of data, 290-292
creating quantile-quantile (Q-Q) plots in other distributions, 319-322
creating scatter plot of multiple groups, 278-280
density function for probability distributions, 228-231
displaying several on one page, 328-331
graphing value of a function, 325-328
graphs from running checkresiduals on ARIMA model, 488
hierarchical cluster dendrogram, 434
histogram, adding density estimate to, 315-316
inserting into R Markdown output document, 526-530
 common output settings, 528
 enlarging output to full page, 528
 shrinking with out.width, 527
multiple datasets in one plot, 305-306
of autocorrelation function of time series, 473-475
of detrended time series, 481
of paired variables, 288-290
of partial autocorrelation function for time series, 476-478
of smoothed time series, 496
of time series data, 452
of time series forecast, 491
plotting a line from x and y points, 300-301
plotting ACF of linear model residuals, 377
plotting linear model object, 371
plotting regression line of a scatter plot, 284-288
plotting regression residuals, 364, 369
plotting TukeyHSD function returns, 385
plotting variables in multiple colors, 322-325
saving to file in ggplot, 331
using function call pipelines, 50
volatility plot for prices, 470
plot_layout function, 328
pmap function, 188, 190
pmax function, 399

pmin function, 399
pnorm function, 225
point plots, 323
poly function, 360
 raw = TRUE parameter, 360
polynomial, regressing on, 360
POSIXct class, 196
POSIXct object, 206
POSIXlt class, 196
POSIXlt object, 208
possibly function, 509
power transformation, finding the best,
 363-368
ppoints function, 319
Practical Regression and ANOVA Using R (Faraway), 437
prcomp function, 429, 432
precedence, operator, 46-47
 for user-defined operators, 414
predicates
 defining for discard function, 152
 in data.frame function, 159
predict function, 378, 437
 interval parameter, 379
predictions
 forming prediction intervals, 379
 predicting a binary-valued variable, 436-438
 predicting new values from linear model,
 378
predictors, 333, 336, 347
 choosing best subset, 353-357
 in linear model formulas, 336
interaction between, plotting, 382-383
multiple, specifying in linear regression formula, 337
removing low-correlation variables from,
 348
presentation, creating from R Markdown document, 546-548
principal component analysis (PCA), 429, 432,
 442
print function
 digits parameter, 88
 redirecting output to file with sink function,
 90
printing
 data in columns, 395
 fewer digits or more digits, 88
 ggplot plots, 527
multiple items with cat function, 28
print function methods for different object classes, 130
results of an assignment, 394
single items with print function, 27
probability, 213-231
 calculating for continuous distributions, 225
 calculating for discrete distributions,
 223-225
converting probabilities to quantiles,
 227-228, 241
counting number of combinations, 215
distributions
 getting help on, 215
 names of, 213
generating combinations, 216
generating random numbers, 217-219
generating random permutation of a vector,
 222
generating random samples, 220
generating random sequences, 221
generating reproducible random numbers,
 219
 plotting a density function, 228-231
proc.time function, 78
profile scripts, 81
 reproducibility issues with, 82
programming, simple, 499-513
 creating a local variable, 505
 creating an anonymous function, 510
 creating collection of reusable functions,
 511
 defining functions, 503-505
 if-else statements, 500-502
 iterating with a loop, 502-503
 protecting against errors, 509
 reindenting code automatically, 513
 signaling errors, 508
 switch function, choosing among multiple
 alternatives, 506
projects, creating in RStudio, 60-62
prop.test function, 247, 248, 257
proportions
 equal, testing groups for, 257
 forming confidence interval for, 248
 testing for a sample, 247
pseudomedian, 246
pure functions, 181
purrr package, 37

compact function, 151
data transformations with, 181
map family of functions, 182
other functions protecting against errors, 510
possibly function, 509
tutorials on, 191

Q

q (quit) function, 10, 78
qexp function, 322
qnorm function, 227
qt function, 320
quantile function, 240
quantile-quantile (Q-Q) plots
 creating non-normal distribution Q-Q plots, 319-322
 creating normal Q-Q plot, 317-319
quantiles
 calculating for a dataset, 240
 converting probabilities to, 227-228
 inverting, 241
quantmod package, 453
quartiles, 310
 calculating for a dataset, 241
questions
 searching mailing list archives for answers to, 22
 submitting to R community, 23-26
--quiet command line option, 414
quietly function, 510

R

R
meaning of data frames to a programmer, 134
technical details for programming, 499
R Data Import/Export guide, 120
R for Data Science (Wickham), 267
R Graphics (Murrell), 263
R Graphics Cookbook (Chang), 263
R Markdown, 417, 515-554
 adding title, author, or date to a document, 518-520
 creating a new document, 516-518
 creating a parameterized report, 548-551
 ecosystem as well as a package, 515
 formatting document text, 520
 generating HTML output, 535

generating Microsoft Word output, 539-545
generating PDF output, 536
generating presentation output, 546-548
inserting math equations into documents, 534
inserting plots into output documents, 526-530
inserting section headings in documents, 521
inserting table of data into a document, 531
inserting tables into documents, 530-531
organizing your workflow, 552
 creating R package for reused logic, 553
 keeping focus on content and source logic, 553
 naming Project directories, 552
 using RStudio Projects, 552
passing information to LaTeX rendering engine, 537
showing output from R code in a document, 523
random numbers, generating, 217-219
 pseudorandom number generation in R, 335
 reproducible random numbers, 219
randomness, checking a sequence for, 250
range function, 186
rate parameter (exponential distribution), 321
rbind function, 160, 163, 173
 inputs as mix of tibbles and data frames, 164
rbinom function, 222
RColorBrewer package, 299
RDBMS systems, R reading from, 120
read.csv function, wrapping with possibly, 509
read.xlsx function, 105
readLines function, 113
readr package, benefits of using, 95
read_csv function, 101-103, 164
 reading CSV data from the web, 104
read_csv2 function, 103
read_fwf function, 95-97
read_html function, 111-113
read_rds function, 125
read_sas function, 110
read_table function, 101
read_table2 function, 97-101
 reading tabular data from the web, 104
records, fixed-width, reading, 94-97
recovering from errors, 509
recursive functions, 504

Recycling Rule, 40, 54
combining pmin and pmax functions with, 400
understanding, 137-139

regression, 333
(see also linear regression)
logistic, 436-438
performing simple orthogonal regression, 430-433

regression coefficients, 336, 407
and autocorrelation of residuals, 377
confidence intervals for, 341
forming confidence intervals for, 368
from linear models, 333
in multiple linear regressions, 338
information about, extracted from linear model, 343
slope, 405

regular expressions, 16
as delimiters in splitting a string, 201
matching files in specific pattern, 91
using in substring substitutions, 202

relative frequencies, 237

removeTable function, 108

rename function, 170

render function, 536

reorder function, 298

rep function, 38

repeat loops, 500

reporting errors, 509

repos option, 75

reprex package, 26

reproducible examples, 23-26

require function, 54, 70

resid function, 481

residuals
checking for ARIMA model, 488
from linear model, 341
in linear model, 480
plotting regression residuals, 369
plotting with ggplot, 364
statistics on regression residuals, 343
testing for autocorrelation, 376-378

response, 333, 336
in linear model formulas, 336
regressing on variables highly correlating with, 347-350

return statements, 504

return values, 499, 504

reusable code snippets, 48

RHOME subcommand, 80

right_join function, 175

rm function, 32

Rmetrics, 196

RMySQL package, 118

rnorm function, 337

rollapply function, 468, 471

rolling mean, 468

rollmean function, 467

rownames attribute, 156

rows
appending to a data frame, 162-165
applying a function to each matrix row, 185-186
applying a function to each row in data frames, 184
initializing a data frame from row data, 160-162
selecting one row from a matrix, 157

rowSums function, 394

rowwise operation, 184

Rscript program, 77

RSeek, 19, 21

RSiteSearch function, 19

RSQLite package, 120

RStudio
changing CRAN mirror, 74
code editing features, 513
community discussion board, 21, 23
creating a new project, 60-62
creating a new R Markdown document, 516
creating and editing R Markdown documents, 516
deleting variables in Environment Pane, 33
exiting, 9
Files pane, 93
getting help in, 12
indenting code blocks in, 513
installing, 4
installing in the cloud, 84
installing packages from CRAN, 72
Knit to HTML, 535
Knit to PDF, 536
Knit to Word, 539
Knit: Presentation, 546
Knitr menu, Knit with Parameters, 549, 550
Projects, 552
R Markdown cheat sheet, 526

R Markdown presentation formats, 546
shortcuts, 48
starting, 5-7
viewer, interactive, 392

RStudio Desktop (see RStudio)
runif function, 217
runs, 251
runs.test function, 250
rvest package, 111
R_HOME environment variable, 80
 R^2 coefficient of determination, 345

S

safely function, 510
safe_read function, 510
sample function, 220, 221
 random permutation of a vector, 222
samples
 comparing locations nonparametrically, 254
 sampling a dataset by selecting every nth element, 398
 testing for same distribution, 260
 testing mean of, 243
 testing sample proportion, 247
sampling with replacement, 220
sapply function, 184
SAS
 meaning of data frames to a user, 134
 reading data from files, 109-111
sas7bdat package, 109
save function, 122
save.image function, 63
saving
 function definitions to file, 512
 images to file in ggplot, 331
 objects in R, 122-124
 on exiting RSpace, 10
 result of previous command, 65
 value in a variable, 29
 workspace in RStudio, 63
scalars, 130
 comparing a vector to a scalar, 40
 operations between vectors and, 44, 139
scale function, 242
scale_fill_brewer function, 299
scan function, 114-118
scatter plots
 creating for paired observations, 267-268
 creating of multiple groups, 278-280

creating one plot for each group, 290-292
of paired variables, 288-290
plotting regression line of, 284-288
scripts
 redirecting output to file with sink function, 90
 running, 76
 running a batch script, 77-79
sd function, 36
search engine, accessing in R documentation, 13, 17
searches
 broadening by using keywords, 17
 finding minimums or maximums for pairwise elements in vectors, 399
 finding relevant functions and packages, 21
search function, 66
searching a vector for particular value and its position, 398
searching R mailing lists, 22
searching supplied documentation, 16-17
searching web for help on R, 18-21
select function, 50, 165, 169, 172
 changing column name with, 171
SELECT statements (SQL), 119
self-describing data, 100
separators
 hyphen as separator between strings, 203
 in string concatenation with paste, 198
seq function, 38, 185
 creating sequence of dates, 210
sequences
 creating sequence of numbers, 38-39
 indexing time series by sequences of dates, 457
 random, generating, 221
seq_along function, 398
Session menu in RStudio, 12
sessionInfo command, 25
set.seed command, 24
set.seed function, 219, 221, 335
setwd command, 59
shading
 background shading in ggplot graphics, 271
 bars of a bar chart, 298-300
 color shading in point plot, 324
shape parameter (aes function), 278
shapiro.test function, 249
Shapiro-Wilk test, 250

shebang line, starting with `#!`, 79
shell prompt, starting R from, 415
shortcuts for R commands, 9
show_query function, 121
significance
 of linear models, 333
 testing a correlation for, 255-256
simulate_pi function, 418, 421
sine wave, damped, 327
sink function, 90
slope (regression coefficient), 405, 432
smoothing a time series, 495-497
sort function, data frames and, 403
sorting
 data frames, 403
 fixing in ggplot bar chart, 293
 in ggplot bar chart with confidence intervals, 296
source function, 76, 81, 511
Spearman method, 255
split function, 422
SQL (Structured Query Language)
 CASE WHEN statements, 192
 in MySQL database queries, 119
meaning of data frames to a programmer, 133
 turning dplyr commands into, 120-122
SQLite database, 120
 loading example data into, 121
 RSSQLite package, 122
sqrt function, 45
SSH, 420
stack function, 141
Stack Overflow website, 2, 20
 post on creating reproducible examples, 26
 submitting questions to, 24
StackExchange website, 20
standard deviation
 calculating, 36-37
 calculating in figures month by month, 470
 in example data for linear regression, 335
 in ggplot graph line, 308
standard error
 for ARIMA model coefficients, 485
 in ggplot bar chart with confidence intervals, 296
 of residuals, 344
 se parameter of ggplot, 284
startup message, suppressing, 414
startup, customizing for R sessions, 81-84
Statistical Analysis Software (see SAS)
statisticians, meaning of data frames to, 133
statistics, 233
 basic, computing, 35-38
 applying a function to each column of data frame, 37
 bootstrapping a statistic, 438-441
 calculating quantiles and quartiles of a data-set, 240
 calculating relative frequencies, 237
 comparing locations of two samples non-parametrically, 254
 comparing means of of two samples, 252-253
 converting data to z-scores, 242
 forming confidence interval for a mean, 244-245
 forming confidence interval for a median, 246
 forming confidence interval for a proportion, 248
 inverting a quantile, 241
 pairwise comparisons between group means, 258-259
 regression statistics, getting, 339-342
 tabulating a factor and creating contingency tables, 238
 testing a sample proportion, 247
 testing categorical variables for independence, 239
 testing correlation for significance, 255-257
 testing for normality, 249
 testing for runs, 250-252
 testing groups for equal proportions, 257
 testing the mean of a sample, 243
 testing two samples for same distribution, 260
stats in ggplot graphs, 266
stats::lag function, 464
stat_function, 326
stat_qq and stat_qq_line functions, 317
stat_summary function, 296
stdout, Rscript output to, 78
step function, 352-357
stepwise regression, 352-357
stftime, 205
stop function, 508
str function, 31

revealing internal structure of variables, 407
strings, 195
concatenating, 198
converting dates to, 205
converting to dates, 204
Date object converted to, 204
extracting substrings, 199
generating all pairwise combinations of, 202
generating combinations of, 216
getting length of, 197
replacing substrings within a string, 201
splitting according to a delimiter, 200
str function, 31
vectors of, returned by ls function, 31
strsplit function, 200
structures, 25, 127
(see also data frames; factors; lists; matrices; tibbles; vectors)
converting between structured types, 178-180
Student's t distribution, 320
sub function, 201
subdirectories, listing files in, 92
sublists in R Markdown, 522
subscripting, matrix-style, 167, 169
subscripts of a vector or list, iterating over, 503
subsets
regressing on, 348, 357
subsetting a time series, 456-458
substr function, 199
summarize function, 191
summarizing data, 235-237
summary function, 71, 235, 240, 397
understanding the regression summary, 342-345
using on linear models, 340
summing rows and columns, 394
suppressMessage function, 410
suppressWarnings function, 410
survival function, 224, 226
switch function, 506
switch statements, 500
Sys.Date function, 204
Sys.getenv function, 80, 415
Sys.putenv function, 415
Sys.setenv function, 81, 415
Sys.time function, 409

T

t statistic, 344, 377
t-test, 243, 252, 254
t.test function, 243, 245, 252
table function, 238, 240
factors as input, 140
tables
ANOVA, of linear regression models, 335
in R Markdown output to Word, 544
inserting computer-generated data into R Markdown document, 531
inserting into R Markdown output document, 530-531
tabular data files
data frames, 133
reading, 97-101
reading from the web, 104
tail function, 391, 454
techniques, advanced
bootstrapping a statistic, 438-441
calculating eigenvalues or eigenvectors, 428
factor analysis, 441-445
finding clusters in data, 433-436
minimizing or maximizing multiparameter functions, 426-428
minimizing or maximizing single-parameter functions, 425
performing principal component analysis, 429
performing simple orthogonal regression, 430-433
predicting a binary-valued variable, 436-438
templates
controlling R Markdown output formatting in Word, 545
for creating R Markdown documents, 518
for R Markdown document output, 519
LaTeX, for formatting documents, 538
text editors, 516
text, formatting in R Markdown documents, 520
theme function, 270
legend.position setting, 281
themes, 266
applying to ggplot graphics, 274-278
tibble function, 159
tibbles, 97, 162
comingling with data frames, 164
creating from vectors in a list, 159

defined, 134
tibble and as_tibble functions, not changing character data, 159
tic function, 408, 418
tidygraph package, installing, 74
tidyrr package, 303
tidyverse packages
 helper functions for calculating basic statistics, 37
 installing, 111
 select function, 166
 tibbles, 97, 134
time series analysis, 447-497
 calculations on time series, 466
 computing moving average of a time series, 467
 computing successive differences, 464-466
 date vs. datetime, 448
 detrending a time series, 479-482
 extracting newest or oldest observations, 454-456
 filling or padding a time series, 460-463
 finding lagged correlations between time series, 478
 fitting ARIMA model to a time series, 482-486
 functions for, 410
 lagging a time series, 463
 making forecasts from ARIMA model, 490
 merging several time series, 458-460
 other representations of data, 448
 packages for, zoo and xts, 447
 plotting a time series forecast, 491
 plotting partial autocorrelation function, 476-478
 plotting the autocorrelation function, 473-475
 plotting time series data, 452
 removing insignificant ARIMA coefficients, 486
 representing time series data, 449-452
 running diagnostics on ARIMA model, 487-490
 subsetting a time series, 456-458
 testing for autocorrelation, 475
 testing for mean reversion, 492-495
Time-Series object types, 494
timeDate package, 196
timeSeries package, 452
timing code, 408-410
tinytex package, 537
titles
 adding to ggplot graphics, 268
 setting for R Markdown document, 519
 setting with ggtitle, 327
toc function, 408, 418
total least squares (TLS), 431
 (see also orthogonal regression)
transformations, 181-193
 applying a function to each column in a matrix or data frame, 186-188
 applying a function to each list element, 182-184
 applying a function to each matrix row, 185, 186
 applying a function to each row in data frames, 184
 applying a function to groups of data, 191
 applying a function to parallel vectors or lists, 188-191
creating new column in a data frame based on a condition, 192-193
regressing on transformed data, 361-363
trends, identifying and removing from a time series
trends, identifying and removing from a time series, 479-482
TRUE and FALSE logical values, 39
ts class, 448
tseries::adf.test function, 17
tsibble package, 448
TukeyHSD function, 383-386

U

union of all dates, 459, 460
unit root tests, 495
Unix
 search path, 67
 shebang line starting with #!, 79
 Sys.getenv function results, 80
unlist function, 150, 412
URLs, using to read data from the web, 104

V

variables
 assigning value to, 29
 categorical, representing with factors, 132
 data type, changing at will, 30

deleting, 32
.Last.value, 65
listing, 31
local, 505
plotting all variables against other variables, 288-290
stripping attributes from, 404
variance, 132
(see also analysis of variance)
calculating, 35
vectorized operations of R, eliminating need for loops, 503
vectors
appending data to, 135
applying a function to parallel vectors or lists, 188-191
arithmetic operations on, 44-45
comparing, 39-41
comparing elements in parallel with pmin and pmax, 399
constructing data frames from, 158
converting between other structured types, 178
converting matrices to, 402
converting to z-scores, 243
creating, 34-35
elements having same mode, 129
finding position of a particular value, 397
flattening a list into, 150
flattening matrices into, 203
formatting with print and format functions, 89
getting length of, 197
inserting data into, 136
key properties of, 127
multiple, combining into one vector and a factor, 140
of atomic types, converting to other types, 177
of levels (see factors)
of names and values, populating list with, 148
of probabilities in quantile functions, 227
of strings, 198
of unequal length, 137-139
of unequal lengths, 54
of years, months, and days converting to Date objects, 207
printing with cat function, 28
random permutation of, 222
results of expressions in R, 8
scalars as one-element vector, 130
selecting elements from, 41
selecting every nth element, 398
summary of, 235
transforming into matrices, 130
turning a vector into a matrix, 155
use in building data frames, 133
versions, extracting for installed packages, 68
viewing data, partial and full dataset, 391
vignettes, 1
listing for a package, 18
openxlsx, 107
readr, 101
sql-translation, 122
volatility, calculating by calendar periods, 470

W

warning function, 509
warnings, suppressing, 410, 526
web search for help on R, 18-21
web, reading CSV and tabular data from, 105
which.max function, 366, 398
which.min function, 398
while loops, 500
wide vs. long data (ggplot), 266, 302
wilcox.test function, 246, 254
Wilcoxon–Mann–Whitney test, 254
window function, 457
selecting range of consecutive dates, 458
Windows
Alt and Ctrl key combinations, 48
Cannot Open File error, 94
exiting RStudio, 9
installing R, 3
starting RStudio, 5
Sys.getenv function results, 80
\ (backslash) in file paths, escaping, 56
Word (see Microsoft Word)
working directory, 59
workspace
indicated by .GlobalEnv, 66
saving, 10, 63
write.xlsx function, 107
writeData function, 108
writeDataTable function, 108
write_csv function, 103
write_rds function, 125

X

xtabs function, 239

xts object, 449

plot function, 453

xts package, 447

Y

YAML header in R Markdown documents, 519

LaTeX options set in, 539

year, month, and day, converting to a date, 206

year/month (yyyymm) subsetting, 458

yyyy-mm-dd format for dates, 205

Z

z- scores, 45

converting data to, 242

zoo object, 449

Date objects as index, 451

plot function, 452

zoo package, 266, 447

About the Authors

J.D. Long is a misplaced southern agricultural economist currently working for Renaissance Re in New York City. J.D. is an avid user of Python, R, AWS, and colorful metaphors, and is a frequent presenter at R conferences as well as the founder of the Chicago R User Group. He lives in Jersey City, NJ, with his wife, a recovering trial lawyer, and his 11-year-old circuit-bending daughter.

Paul Teator is a quantitative developer with master's degrees in statistics and computer science. He specializes in analytics and software engineering for investment management, securities trading, and risk management. He works with hedge funds, market makers, and portfolio managers in the greater Chicago area.

Colophon

The animal on the cover of *R Cookbook* is a harpy eagle (*Harpia harpyja*). One of the 50 species of eagle in the world, the harpy eagle is native to the tropical rain forests of Central and South America, and prefers to nest in the upper canopy layer thereof. Both its genus and species names refer to the harpies of ancient Greek mythology—vicious creatures with the face of a woman and the body of an eagle or vulture.

On average, harpy eagles weigh about 18 lbs, are 36 to 40 inches long, and have a wingspan of 6 to 7 feet, though females are consistently larger than males. The plumage of both sexes is identical, however: slate-black feathers dominate the animal's top half, while the underside is white or light gray. Light gray-colored heads are accentuated with a double crest of large feathers, which specimens can raise when showing hostility.

Harpy eagles are monogamous, and pairs raise only one chick every two to three years. Females will usually lay two eggs at a time, and after the first hatches, the other is neglected and does not hatch. Though the chick will fledge within six months, both parents continue to care for and feed the chick for at least a year. Because of this low rate of population growth, the harpy eagle is particularly susceptible to encroachments on its habitat and losses from human hunting. Throughout its range, the animal's conservation status ranges from threatened to critically endangered.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from J.G. Wood's *Animate Creation*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.