Portland State University

PDXScholar

TREC Final Reports

Transportation Research and Education Center
(TREC)

1-2017

Introduction to Data Science for Transportation Researchers, Planners, and Engineers

Liming Wang

Portland State University, limwang@pdx.edu

Follow this and additional works at: https://pdxscholar.library.pdx.edu/trec_reports

Part of the [Transportation Commons](#), [Urban Studies Commons](#), and the [Urban Studies and Planning Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Wang, Liming. Introduction to Data Science for Transportation Researchers, Planners, and Engineers. NITC-ED-854. Portland, OR: Transportation Research and Education Center (TREC), 2018. <https://doi.org/10.15760/trec.192>

This Report is brought to you for free and open access. It has been accepted for inclusion in TREC Final Reports by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

FINAL REPORT

Introduction to Data Science for Transportation Researchers, Planners, and Engineers

NITC-ED-854 ■ January 2018

*NITC is a U.S. Department of Transportation
national university transportation center.*

INTRODUCTION TO DATA SCIENCE FOR TRANSPORTATION RESEARCHERS, PLANNERS, AND ENGINEERS

Final Report

NITC-ED-854

by

Liming Wang
Portland State University

for

National Institute for Transportation and Communities (NITC)
P.O. Box 751
Portland, OR 97207

January 2017

Technical Report Documentation Page

1. Report No. NITC-ED-854	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle Introduction to Data Science for Transportation Researchers, Planners, and Engineers		5. Report Date January 2017	
		6. Performing Organization Code	
7. Author(s) Liming Wang		8. Performing Organization Report No.	
9. Performing Organization Name and Address Portland State University 506 SW Mill St. Portland, OR 97207-0751		10. Work Unit No. (TRAIS)	
		11. Contract or Grant No. 854	
12. Sponsoring Agency Name and Address National Institute for Transportation and Communities (NITC) P.O. Box 751 Portland, OR 97207		13. Type of Report and Period Covered	
		14. Sponsoring Agency Code	
15. Supplementary Notes			
16. Abstract <p>Building on the successful scientific computing training program offered by the Software Carpentry (http://www.software-carpentry.org/), this course exposes students to the best practices in data science through hands-on lab sessions. Using transportation data and examples, it also aims to help students tackle the challenge of "drinking from a hose" when dealing with the overwhelming amount of data that is increasingly common in transportation research and practice.</p> <p>Although computing is now an integral part of every aspect of science and engineering, transportation research included, most students of science, engineering, and planning are never taught how to build, use, validate, and share software well. As a result, many spend hours or days doing things badly that could be done well in just a few minutes.</p> <p>The goal of this course is to help students gain the necessary skills to spend less time wrestling with software and more time doing useful research.</p>			
17. Key Words Data science; best practices; reproducible research		18. Distribution Statement No restrictions. Copies available from NITC: http://nitc.trec.pdx.edu/research/project/854	
19. Security Classification (of this report) Unclassified	20. Security Classification (of this page) Unclassified	21. No. of Pages 118	22. Price

ACKNOWLEDGMENTS

This course was developed with financial support from the National Institute of Transportation and Communities under grant number 854.

I am grateful to the participants of the 2017 summer course for their valuable feedback. The support of TREC staff, in particular, Lisa Patterson and Eva-Maria Muecke, is greatly appreciated. Jamaal Green and Dillon Mahmoudi, both graduate students at Portland State University, have given me valuable feedback and assistance in the process of developing the course.

Parts of the course materials have been adapted from the following sources:

- [R for Data Science](#) by Hadley Wickham
- [Software Carpentry workshop lessons](#)
- [UBC Stat 545](#) by Professor Jenny Bryan at UBC
- [NEU 5110 Introduction to Data Science](#) by Professor Jan Vitek

The write-up and website are powered by the [bookdown](#) package and [GitHub](#).

DISCLAIMER

The contents of this report reflect the views of the authors, who are solely responsible for the facts and the accuracy of the material and information presented herein. This document is disseminated under the sponsorship of the U.S. Department of Transportation University Transportation Centers Program in the interest of information exchange. The U.S. Government assumes no liability for the contents or use thereof. The contents do not necessarily reflect the official views of the U.S. Government. This report does not constitute a standard, specification or regulation.

RECOMMENDED CITATION

Wang, Liming. *Introduction to Data Science for Transportation Researchers, Planners, and Engineers*. NITC-ED-854. Portland, OR: Transportation Research and Education Center (TREC), 2018.

TABLE OF CONTENTS

EXECUTIVE SUMMARY	7
1.0 SYLLABUS	8
1.1 FORMAT, SOFTWARE AND HARDWARE.....	8
1.2 PREREQUISITE.....	8
1.3 TEXTBOOK AND READINGS	8
1.4 TOPICS	9
1.5 LICENSE	9
1.6 ACKNOWLEDGEMENTS	9
2.0 OVERVIEW AND INTRODUCTION	11
2.1 PART I: DATA SCIENCE BEST PRACTICES	11
2.2 SET UP YOUR COMPUTER	11
2.2.1 Installation.....	11
2.2.2 Installation Verification	11
2.3 WHY R	11
2.4 INTRODUCTION TO R AND RSTUDIO	11
2.4.1 Project Organization in RStudio	12
2.5 EXERCISE	12
2.6 CLASS PROJECT	12
2.7 LEARNING MORE.....	13
3.0 R CODING BASICS.....	14
3.1 READINGS	14
3.2 BEST PRACTICES FOR DATA SCIENCE.....	14
3.3 R CODING BASICS	15
3.4 ADVANCED TOPICS	15
3.4.1 Code Style Guide	15
3.4.2 R Command-Line Program.....	15
3.4.3 Debugging with RStudio.....	17
3.5 EXERCISE	17
3.6 LEARNING MORE.....	17
4.0 VERSION CONTROL WITH GIT	18
4.1 READINGS	18
4.2 INSTALL GIT	18
4.2.1 For Windows.....	18
4.2.2 For Mac	18
4.2.3 For Linux	18
4.3 LESSON	18
4.4 EXERCISE	19
4.5 LEARNING MORE.....	19
5.0 CREATE R PACKAGE	20
5.1 READINGS	20

5.2	LESSON	20
5.3	R PACKAGE CHECKLIST	20
5.4	EXERCISE	20
5.5	LEARNING MORE.....	20
6.0	R MARKDOWN	21
6.1	READINGS	21
6.2	LESSONS	21
6.2.1	Formatting Text in Markdown.....	21
6.2.2	Use knitr to Produce a Report.....	22
6.3	EXERCISE	22
6.4	LEARNING MORE.....	22
7.0	PART II: OVERVIEW AND INTRODUCTION.....	23
7.1	PART II: TIDYVERSE WORK FLOW	23
7.2	READINGS	23
7.3	THE TIDYVERSE WORKFLOW	23
7.4	THE TIDYVERSE PACKAGES	24
7.5	PIPE OPERATOR	24
7.5.1	RStudio Keyboard Shortcut for %>%	25
7.6	CLASS PROJECT	25
8.0	DATA IMPORTING	27
8.1	READINGS	27
8.2	TIBBLES VS. DATA.FRAME	27
8.2.1	Data Frames are Awesome	27
8.2.2	Printing.....	28
8.2.3	Subsetting.....	29
8.2.4	Converting.....	29
8.3	DATA IMPORT	30
8.4	EXERCISE	32
8.5	LEARNING MORE.....	33
9.0	TIDY DATA	34
9.1	READINGS	34
9.2	TIDY DATA.....	34
9.3	LESSON	34
9.3.1	Spreading and Gathering.....	36
9.3.1.1	Gathering	36
9.3.1.2	Spreading	38
9.3.2	Separating and Uniting	39
9.3.2.1	Separate	40
9.3.2.2	Unite.....	42
9.4	EXERCISE	43
9.5	LEARNING MORE.....	44
10.0	DATA MANIPULATION WITH DPLYR.....	45
10.1	READINGS	45
10.2	OVERVIEW	45
10.3	LESSON	45

10.3.1	Prerequisites	45
10.3.2	Use filter() to Subset Data Row-wise.	45
10.3.3	Use select() to Subset the Data on Variables or Columns.	46
10.3.4	Revel in the Convenience	47
10.3.5	Pure, Predictable, Pipeable	47
10.3.6	Use mutate() to Add New Variables	48
10.3.7	Use arrange() to Row-order Data in a Principled Way	51
10.3.8	Use rename() to Rename Variables	52
10.3.9	select() Can Rename and Reposition Variables	53
10.3.10	group_by() is a Mighty Weapon	53
10.3.10.1	Counting Things Up	54
10.3.10.2	General Summarization	55
10.3.11	Grouped Mutate	57
10.3.11.1	Computing With Group-wise Summaries	57
10.3.11.2	Window Functions	58
10.3.12	Grand Finale	61
10.3.13	Working With Two Datasets	61
10.3.14	Variable Recoding With dplyr	62
10.3.15	Programming With dplyr	64
10.4	EXERCISE	64
10.5	LEARNING MORE	65
11.0	DATA VISUALIZATION WITH GGLOT2	66
11.1	READINGS	66
11.2	OVERVIEW	66
11.3	TIPS	66
11.4	LESSON	66
11.4.1	Prerequisites	66
11.4.2	First Steps	66
11.4.3	The mpg Data Frame	67
11.4.4	Creating a ggplot	67
11.4.5	A Graphing Template	69
11.4.6	Aesthetic Mappings	69
11.4.7	Facets	73
11.4.8	Geometric Objects	75
11.4.9	Statistical Transformations	81
11.4.10	Position Adjustments	87
11.4.11	Coordinate Systems	94
11.4.12	The Layered Grammar of Graphics	97
11.4.13	Additional Resources	99
11.5	EXERCISE	99
11.6	LEARNING MORE:	100
12.0	SPLIT-APPLY-COMBINE PATTERN IN DATA MODELING	101
12.1	READINGS	101
12.2	SPLIT-APPLY-COMBINE	101
12.3	LESSON	101
12.3.1	Think Before You Create Excerpts of Your Data	101

12.3.2 Prerequisite	102
12.3.3 Split-Apply-Combine.....	102
12.3.4 Entry Level: dplyr::group_by().....	103
12.3.5 General Approach: Nesting.....	103
12.3.6 Apply a Function purrr::map() and mutate()	105
12.3.7 Always Have an Exit Strategy	107
12.3.8 Simplify and Combine	107
12.3.9 Recap.....	108
12.3.10 Enjoy the Payoff of Your Work.....	109
12.4 EXERCISE	112
12.5 RESOURCES:	112
13.0 WHERE TO GO FROM HERE.....	113
13.1 NOTICEABLE FEATURES NOT COVERED	113
13.1.1 Automate Testing Your Code	113
13.1.2 Automate Your Workflow	113
13.1.3 Develop Interactive Apps With Shiny	113
13.2 WHERE TO FIND HELP.....	114
REFERENCES.....	115

LIST OF TABLES

Table 9.1 Bike counts by day of week and bridge (1st format).....	43
Table 9.2 Bike counts by day of week and bridge (2nd format).....	43

LIST OF FIGURES

Figure 9.1 Gathering table4 into a tidy form	37
Figure 9.2 Spreading table2 makes it tidy	39
Figure 9.3 Separating table3 makes it tidy.....	41
Figure 9.4 Uniting table5 makes it tidy	42
Figure 11.1 Scatter plot of highway mpg by displ.....	68
Figure 11.2 Scatter plot with some cars highlighted in red	69
Figure 11.3 Scatter plot of highway mpg by displ, with class of cars differentiated using color. 70	
Figure 11.4 Scatter plot of highway mpg by displ, with class of cars differentiated using size... 71	
Figure 11.4 Scatter plot of highway mpg by displ, with class of cars differentiated using size... 72	
Figure 11.5 Scatter plot with color aesthetic manually set to blue	73
Figure 11.6 Scatter plots of highway mpg by displ, faceted by class	74
Figure 11.6 Scatter plots of highway mpg by displ, faceted by class	75
Figure 11.7 Scatter plot and smooth curve of highway mpg by displ	76
Figure 11.8 Smooth curves of highway mpg by displ, grouped by drv	78
Figure 11.9 Scatter plot and smooth curves of highway mpg by displ, grouped by drv	79
Figure 11.10 Smooth curves of highway mpg by displ	80
Figure 11.11 Scatter plot of highway mpg by displ super-imposed with stat_smooth.....	81
Figure 11.12 Bar chart of diamonds by cut.....	82

Figure 11.13 Process behind geom_bar	83
Figure 11.14 Bar chart of diamonds by cut created with stat_count	84
Figure 11.15 Bar chart created with stat=identity	85
Figure 11.16 Bar chart with y-axis in proportion	86
Figure 11.17 Chart with stat-summary	87
Figure 11.18 Bar chart with fill and color.....	88
Figure 11.19 Stacked bar chart of diamonds by cut and clarity.....	89
Figure 11.20 Bar chart with position='identity'	90
Figure 11.21 Bar chart with position='fill'	91
Figure 11.22 Bar chart with position='identity'	92
Figure 11.23 Scatter plot of highway mpg by displ.....	93
Figure 11.24 Scatter plot of highway mpg by displ, with position='jitter'.....	94
Figure 11.25 Bar chart with flipped x and y axis.....	95
Figure 11.26 Map of New Zealand with ggplot2.....	96
Figure 11.27 Coxcomb chart with the flipped and polar coordinate system	97
Figure 11.28 Visualization Grammar Step 1	98
Figure 11.29 Visualization Grammar Step 2	98
Figure 11.30 Visualization Grammar Step 3	99
Figure 12.1 Distributions of model intercepts and slopes and their relationship in a scatter plot	111

EXECUTIVE SUMMARY

With almost every aspect of transportation research and practice driven to utilize complex computer software and innovative data sources (University Transportation Research Center, 2014), researchers and professionals increasingly face the computing challenges that have plagued other science and engineering disciplines. Most students of science, engineering, and planning are never taught to build, use, validate, and share software well (Merali, 2010). To help students and professionals in transportation research cope with these challenges, this project will apply lessons from similar programs in other disciplines and aims to equip them with proper scientific computing skills.

The direct outcomes of this project are a course plan and materials for training transportation students and professionals in basic data science. The course materials are open source under the Creative Commons (CC) license and publicly available online on GitHub at <https://cities.github.com/datascience>. Any school or instructor can replicate the course. The course would also be helpful to transportation practitioners who may use it for self-instruction.

The intangible outcomes of this project will be groups of transportation students who are better equipped for transportation research in the age of computing, and who can spend less time wrestling with software and more time doing useful research.

The project identifies and absorbs the best practices in data science in the academic and professional literature (for example, G. Wilson et al., 2014) as well as successful courses and workshops for similar purposes, such as the Software Carpentry lessons, and develops the following course topics:

- Best practices in data science
- Coding and scripting basics
- Version control using Git
- The import-tidy-transform-visualize-model-communicate workflow
- Communication and reproducible research
- How to find help

The course was first offered as a [Transportation Research and Education Consortium \(TREC\) summer course in 2017](#). Future offerings of the course are being planned.

1.0 SYLLABUS

Did you ever feel you are “drinking from a hose” with the amount of data you are attempting to analyze? Have you been frustrated with the tedious steps in your data processing and analysis process and thinking, “There’s gotta be a better way to do things”? Are you curious what the buzz of data science is about? If any of your answers are yes, then this course is for you.

Although computing is now integral to every aspect of science and engineering, transportation research included, most students of science, engineering, and planning are never taught how to build, use, validate, and share software well. As a result, many spend hours or days doing things badly that could be done well in just a few minutes and in a repeatable and self-documented way. The goal of this course is to empower students to spend less time wrestling with software and more time doing useful research/work.

This course builds on successful data science training programs, such as the Software Carpentry (<http://www.software-carpentry.org/>) and Data Carpentry, and recent developments in related software and research. It exposes transportation students and professional to the best practices in data science and scientific computing through lectures, discussion, and hands-on lab sessions, and aims to help them tackle the challenge of “drinking from a hose” when dealing with an overwhelming amount of data.

1.1 FORMAT, SOFTWARE AND HARDWARE

Classes will all be hands-on sessions with lectures, discussions, and labs. A major component of the class is the class project, in which students go through actual data processing, analysis and reporting while learning the best practices of data science.

This course will use the free statistical software R, with RStudio (<https://www.rstudio.com/>) as the main interface. The lecture and lab instructions will use R. It is possible (and encouraged) for existing Python users (and potentially users of other software, such as Stata, Matlab, etc.) to keep using the software they already know well. Students are encouraged to bring their own laptops. The instructor and TA will help the students set laptops up to run all examples and exercises in lectures and labs, and to re-run them later for review and their own project.

1.2 PREREQUISITE

Basic knowledge and experience of working with quantitative data; experiences and skills in (or keenness to learn) a programming language (e.g., Python); and/or data processing and statistical software (e.g., R, Matlab, Stata).

1.3 TEXTBOOK AND READINGS

The course will use the following textbook:

- Wickham, H., Grolemund, G., 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*, 1 edition. ed. O'Reilly Media. (R4DS)

An electronic version is available on [Hadley Wickham's website](#).

For Python users, Wes McKinney's book is recommended:

- [McKinney, W., 2012. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, 1 edition. ed. O'Reilly Media.](#)

Journal articles and online resources are used as supplements to the textbook.

1.4 TOPICS

Topics are tentative and subject to change according to students' needs, but will be centered on these topics:

- Part I:
 1. [Overview and Introduction](#)
 2. [R Coding Basics](#)
 3. [Version Control with Git](#)
 4. [Create R Package](#)
 5. [Producing Reports with R Markdown](#)
- Part II:
 1. [Data Importing](#)
 2. [Tidy Data](#)
 3. [Data Manipulation with dplyr](#)
 4. [Data Visualization with ggplot2](#)
 5. [Split-Apply-Combine Pattern in Data Modeling](#)
 6. [Where To Go From Here](#)

1.5 LICENSE

The course materials are made available under the [Creative Commons Attribution license](#).

1.6 ACKNOWLEDGEMENTS

This course is developed with financial support from the National Institute of Transportation and Communities for project #854.

I am grateful to the participants of the 2017 summer course for their valuable feedback. The support of TREC staff, in particular, Lisa Patterson and Eva-Maria Muecke, is greatly appreciated. Jamaal Green and Dillon Mahmoudi, both graduate students at Portland State University, have given me valuable feedback and assistance in the process of developing the course.

Parts of the course materials have been adapted from the following sources:

- [R for Data Science](#) by Hadley Wickham
- [Software Carpentry workshop lessons](#)
- [UBC Stat 545](#) by Professor Jenny Bryan at UBC
- [NEU 5110 Introduction to Data Science](#) by Professor Jan Vitek

The writeup and website are powered by the [bookdown](#) package and [GitHub](#).

2.0 OVERVIEW AND INTRODUCTION

2.1 PART I: DATA SCIENCE BEST PRACTICES

1. [Overview and Introduction](#)
2. [R Coding Basics](#)
3. [Version Control with Git](#)
4. [Create R Package](#)
5. [Producing Reports with R Markdown](#)

2.2 SET UP YOUR COMPUTER

2.2.1 Installation

- Install R 3.4.1 from <https://ftp.osuosl.org/pub/cran/>. (Requires administrator privileges)
- Install RStudio Desktop free version from <https://www.rstudio.com/products/rstudio/download/>
- Git: Follow [these steps](#) to install Git for Windows (or for the specific operating system on your laptop). (Requires administrator privileges)

2.2.2 Installation Verification

1. Launch RStudio and you should see a program window like this:
2. Click the **File** menu, select **New Project...**, then **Version Control** and **Git**;
3. Copy & paste this URL: <https://github.com/cities/datascience2017.git> into the Repository URL textbox;
4. Click **Create Project**.

If you see a popup box that says "Clone Repository" with a progress bar and then RStudio refreshes, then your installation is working.

2.3 WHY R

- [Free, as in beer and speech](#)
- Large and growing community, with [more than 13,000 packages and growing](#)
- Powerful and flexible
 - [An \(incomplete\) list of models implemented as R packages](#)
 - [Graphics Gallery](#)
 - [Interactive Web Apps: Interactive Plots, Dashboard, Widget ...](#)

2.4 INTRODUCTION TO R AND RSTUDIO

1. [Introduction to R and RStudio](#)
2. [Seeking help](#)

2.4.1 Project Organization in RStudio

R experts keep all the files associated with a project together — input data, R scripts, analytical results, and figures. This is such a wise and common practice that RStudio has built-in support for this via projects. For example, a project can be an R package under development, or code, data, and documents for a research project, or a book project (for example, Hadley use an RStudio project for the R for Data Science book at <https://github.com/hadley/r4ds>).

An RStudio project must contain a file with extension Rproj and can contain any files within the directory where the project file lives or its subdirectories. Typical types of files in an RStudio project include R Script - R Markdown, R notebook - R Presentation - Shiny Web App - Text file, C++, HTML, ...

Hadley recommends the following for RStudio projects workflow:

- Create an RStudio project for each data analysis project.
- Keep data files there and load them into R with data import.
- Keep scripts there; edit them, run them in bits or as a whole.
- Save your outputs (plots and cleaned data) there.
- Only ever use relative paths (relative to the root path of the project), not absolute paths.

Everything you need is in one place and cleanly separated from all the other projects that you are working on.

I recommend organizing files in subdirectories by types, typically with at least these items: - project_name.Rproj, RStudio project file - README.md, a description file of the project - code, for R scripts - data, for data files - docs, for document/report files, such as R Markdown files

2.5 EXERCISE

Create an RStudio project for this class on your laptop, set up the directory structure as recommended above, download [the bike counts data file](#), and save them in the data directory of your RStudio project.

2.6 CLASS PROJECT

For the class project, you are expected to create an R package with the following requirements and commit it to GitHub:

1. Contains at least one self-contained function;
2. Completed with necessary documentation;
3. Passes R package check and test(s); and
4. [Advanced] includes a vignette that demonstrates the usage of the package.

You can take and/or re-organize code from your current work or start from scratch. Take the feasibility of completing it in a week into consideration when selecting project ideas.

If you don't have a feasible project idea at the moment, consider writing an R package that reads and visualizes the bike counts on Hawthorne Bridge and Tilikum Crossing. Daily traffic counts data for these two bridges can be found [here](#). At the minimum, your package should be able to:

- Read the data in the excel files;
- Process (tidy) data as necessary;
- Visualize bike counts on either or both bridges based on the data frame passed in;
- Plot daily bike counts for any specified period;
- [Advanced] Plot daily, weekly, or monthly bike counts based on a frequency argument;
- Pull weather data for Portland and study (visualize and/or model) the effect of weather on bike counts; and
- [Advanced] Study (visualize and/or model) the effect of Tilikum Crossing (opened in September 2015) on bike counts on Hawthorne Bridge.

2.7 LEARNING MORE

The above will get your basic setup ready but here are some links if you are interested in reading a bit further.

- RStudio Cheat Sheet
 - <https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf>
- RStudio's leads for getting help with R
 - <https://support.rstudio.com/hc/en-us/articles/200717153-Getting-Help-with-R>
- R FAQ:
 - <http://cran.r-project.org/doc/FAQ/R-FAQ.html>
- R Installation and Administration
 - <http://cran.r-project.org/doc/manuals/R-admin.html>
- More about add-on packages in the R Installation and Administration Manual
 - https://cran.r-project.org/doc/manuals/R-admin.html#Add_002don-packages

3.0 R CODING BASICS

3.1 READINGS

1. Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., ... Wilson, P. (2014). Best Practices for Scientific Computing. PLOS Biology, 12(1), e1001745. <https://doi.org/10.1371/journal.pbio.1001745>
2. R for Reproducible Scientific Analysis by Software Carpentry

3.2 BEST PRACTICES FOR DATA SCIENCE

These are the best practices for data science recommended by G. Wilson et al. (2014):

1. Write programs for people, not computers
 - i. a program should not require its readers to hold more than a handful of facts in memory at once
 - ii. make names consistent, distinctive, and meaningful
 - iii. make code style and formatting consistent
2. Let the computer do the work
 - i. make the computer repeat tasks
 - ii. save recent commands in a file for re-use
 - iii. use a build tool to automate workflows
3. Make incremental changes
 - i. work in small steps with frequent feedback and course correction
 - ii. use a version control system
 - iii. put everything that has been created manually in version control
4. Don't repeat yourself (or others)
 - i. every piece of data must have a single authoritative representation in the system
 - ii. modularize code rather than copying and pasting
 - iii. re-use code instead of rewriting it
5. Plan for mistakes
 - i. add assertions to programs to check their operation
 - ii. use an off-the-shelf unit testing library
 - iii. turn bugs into test cases
 - iv. use a symbolic debugger
6. Optimize software only after it works correctly
 - i. use a profiler to identify bottlenecks
 - ii. write code in the highest-level language possible
7. Document design and purpose, not mechanics
 - i. document interfaces and reasons, not implementations
 - ii. refactor code in preference to explaining how it works
 - iii. embed the documentation for a piece of software in that software
8. Collaborate
 - i. use pre-merge code reviews
 - ii. use pair programming when bringing someone new up to speed and when tackling particularly tricky problems
 - iii. use an issue tracking tool

In small groups, each picks (or is assigned) a practice and discusses:

1. What is the current practice? Why it is not a good practice and why is the best practice better?
2. How can we move the current practice to the best practice?

3.3 R CODING BASICS

This section assumes that students know little about R and gets them up to speed with the basics:

1. [Data Structures](#)
 - How can I read data in R?
 - What are the basic data types in R?
 - How do I represent categorical information in R?
2. [Exploring Data Frames](#)
 - How can I manipulate a data frame?
3. [Subsetting Data](#)
 - How can I work with subsets of data in R?
4. [Control Flow](#)
 - How can I work with subsets of data in R?
5. [Visualization with ggplot2](#)
 - How can I create publication-quality graphics in R?
6. [Vectorization](#)
 - How can I operate on all the elements of a vector at once?
7. [Functions Explained](#)
 - How can I write a new function in R?
8. [Writing Good Software](#)
 - How can I write software that other people can use?

3.4 ADVANCED TOPICS

3.4.1 Code Style Guide

In programming, as in writing, it is generally a good idea to stick to a consistent coding style. There are two style guides that you can adopt or customize to create your own:

- [Google's R style guide](#)
- [Hadley Wickham's code style guide](#)

3.4.2 R Command-Line Program

RStudio is good for writing and testing your R code, but for work that needs repetition or takes a long time to finish, it may be easier to run your program/script in command line instead.

Before we start, open the RStudio project you created following [the RStudio project organization recommendations](#) in the Overview section (assuming you downloaded and saved the bike counts data to the data directory of the project).

We can create an R script (from the **File/New File/R Script** menu of RStudio) that loads the bike counts for the Hawthorne Bridge:

```
library(tidyverse)

input_file <- "data/Hawthorne Bridge daily bike counts 2012-2016 082117.xlsx"
bridge_name <- "Hawthorne"
bikecounts <- read_excel(input_file)
names(bikecounts) <- c("date", "westbound", "eastbound", "total")
bikecounts$bridge <- bridge_name

head(bikecounts)
```

Choose a file name, for example, load_data.R, and save the script in the code directory of your RStudio project (create a code directory first if you haven't yet).

Now we can run the script in a command line shell (you can open one in **RStudio's Tools/Shell...** menu):

```
Rscript code/load_data.R
```

Notice that the script may not print out outputs on the screen when called in the command line unless you explicitly call the print function.

But what if we have many files for which we would like to repeatedly show the basic information (rows, data types, etc.)? We can refactor our script to accept the file name and bridge name from command line arguments so that the script can work with any acceptable files.

In an R script, you can use the commandArgs function to get the command line arguments:

```
args <- commandArgs()
print(args)
```

So, in our case, our script should take input_file and bridge_name from the command line arguments, we can get the value of the arguments with:

```
args <- commandArgs()
input_file <- args[1]
bridge_name <- args[2]
```

Replace the two lines in load_data.R starting with input_file and bridge_name with these three lines.

Now our script can be invoked in the command line with:

```
Rscript code/load_data.R "data/Hawthorne Bridge daily bike counts 2012-2016 082117.xlsx" \
Hawthorne
```

(The quotation marks are needed for the file name when there are spaces in the name and "\" breaks a command into two lines.)

3.4.3 Debugging with RStudio

This section is adapted from [Visual Debugging with RStudio](#).

1. Download foo.R from <https://raw.githubusercontent.com/cities/datascience2017/master/code/foo.R> and save it to the code directory of your project folder;
2. Open foo.R and source it;
3. In the RStudio Console pane of type `foo("-1")` and then enter.

Why does the foo function claim "-1 is larger than 0"? Let's debug the foo function and find out.

3.5 EXERCISE

1. Write a function that takes the name of a bike counts data file as input and return a data frame;
 - use the readxl package to read data in excel files
2. Create an R script that utilizes your function to read in data in the Tilikum and Hawthorne bike count files;
3. Do quick summaries of the data for each bridge:
 - How many days of data are there for each bridge?
 - What are the average daily bike counts for each bridge? Minimum? Maximum?
 - What are the average weekly, monthly, and annual bike counts for each bridge?
4. [Advanced] Write a function that calculates average daily, weekly, or monthly bike counts for each bridge based on a frequency argument.

3.6 LEARNING MORE

- [Introduction to R on Data Camp](#): A self-instruction course covering R basics.
- [Try R by Code School](#)

4.0 VERSION CONTROL WITH GIT

4.1 READINGS

- [Happy Git and GitHub for the useR](#)

4.2 INSTALL GIT

4.2.1 For Windows

1. Download the Git for Windows installer.
2. Run the installer and follow the steps below:
 1. Click "Next".
 2. Click "Next".
 3. Keep **"Use Git from the Windows Command Prompt"** selected and click "Next". If you forgot to do this, rerun the installer and select the appropriate option.
 4. Click "Next".
 5. Keep **"Checkout Windows-style, commit Unix-style line endings"** selected and click "Next".
 6. Keep **"Use Windows' default console window"** selected and click "Next".
 7. Click on "Install".
 8. Click "Finish".
3. If your "HOME" environment variable is not set (or you don't know what this is):
 1. Open command prompt (Open Start Menu then type cmd and press [Enter])
 2. Type the following line into the command prompt window exactly as shown: setx HOME "%USERPROFILE%"
 3. Press [Enter], you should see SUCCESS: Specified value was saved.
 4. Quit command prompt by typing exit then pressing [Enter].

4.2.2 For Mac

Install [git-osx-installer](#).

4.2.3 For Linux

install git from your Linux distro's package manager.

4.3 LESSON

Adapted from the Software Carpentry lesson [Version Control with Git](#) and Professor Jenny Bryan's [Happy Git with R](#).

- [A quick overview of Version Control](#);
- Register a Github account at <https://github.com>;
- Use your GitHub username and email to config Git installation on your own computer. Open a Shell/Command line window from RStudio menu **Tools/Shell...**, and modify and run the following commands (of course, replace my name and email with your own):


```
git config --global user.name "Liming Wang"  
git config --global user.email "lmwang@gmail.com"  
git config --global color.ui "auto"
```

- [Cache credentials](#)
- [Collaborating](#)
- [Conflicts](#)
- [Review of common git commands](#)
- [Using Git fro RStudio](#)

4.4 EXERCISE

1. Set up your Git installation for use with GitHub;
2. Turn the RStudio project you have been working on into a Git repository;
3. Commit it to GitHub by following [these steps](#).

4.5 LEARNING MORE

1. [Happy Git with R](#)
2. Software Carpentry's [Version Control with Git](#)
3. [Try Git](#)
4. [Pro Git](#), a book by Scott Chacon and Ben Straub

5.0 CREATE R PACKAGE

5.1 READINGS

- [R packages](#) Getting Started

5.2 LESSON

1. [Developing R Packages with RStudio](#)
2. [Building, Testing, and Distributing Packages](#)
3. [Writing Package Documentation](#)
4. [Writing Documents for R Package](#)

5.3 R PACKAGE CHECKLIST

0. Check out (clone) the package repository as an RStudio project or create an R package project from RStudio menu **File/New Project...**;
1. Test that the package builds and loads without issue;
2. Check the package with `devtools::check()`, which also runs tests included in the package; verify there are no errors or warnings;
3. Does the package include tests? Do the tests run successfully?
4. Verify that the package includes proper documents as roxygen comments in the R code files;
5. Does the package include vignettes?
6. Install the package and load it with `library`;
7. See what functions are provided by the package with `ls("package:<package name>")` after loading it with `library`; and
8. Study the vignettes and help documents to learn how to use it.

5.4 EXERCISE

Reorganize the function(s) you developed in the [R Coding Basics](#) section into an R package:

- Use roxygen comments to specify the dependencies of your function(s);
- Use roxygen comments to document your function(s);
- [Advanced] Create tests for your functions;
- Make sure your package builds and loads properly;
- Go through the R package checklist; and
- Commit your package as a GitHub repository.

5.5 LEARNING MORE

1. [R Package by Hadley Wickham](#)
2. [Making Your First R Package by Fong Chun Chan](#)
3. [A Quickstart Guide for Building Your First R Package](#)

6.0 R MARKDOWN

R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both

- save and execute code and
- generate high-quality reports that can be shared with an audience.

R Markdown documents are fully reproducible and support dozens of static and dynamic output formats.

You write R Markdown documents in a simple plain text-like format, which allows you to format text using intuitive notations so that you can focus on the content you're writing. But you still get a well-formatted document out. In fact, you can turn your plain text (and R code and results) into an HTML file or, if you have an installation of LaTeX and Pandoc on your machine, a pdf, or even a Word document (if you must!).

6.1 READINGS

1. R4DS [Chapter 27](#), [Chapter 30](#), [Chapter 29](#)
2. [R Markdown gallery](#)

6.2 LESSONS

To get started, install the knitr package.

```
install.packages("knitr")
```

When you click on File -> New File, there is an option for "R Markdown...". Choose this and accept the default options in the dialog box that follows (but note that you can also create presentations this way). Save the file and click on the "Knit HTML" button at the top of the script. Compare the output to the source.

6.2.1 Formatting Text in Markdown

Visit http://rmarkdown.rstudio.com/authoring_basics.html and briefly check out some of the formatting options.

In the example document add

- Headers using #
- Emphasis using asterisks: **italics** and ****bold****
- Lists using * and numbered lists using 1., 2., etc.

Markdown also supports LaTeX equation editing. We can display pretty equations by enclosing them in \$, e.g., $\alpha = \frac{1}{(1-\beta)^2}$ renders as: $\alpha = \frac{1}{(1-\beta)^2}$.

The top of the source (.Rmd) file has some header material in YAML format (enclosed by triple dashes). Some of this gets displayed in the output header; some provides formatting information to the conversion engine.

To distinguish R code from text, RMarkdown uses three back-ticks followed by {r} to distinguish a "code chunk". In RStudio, the keyboard shortcut to create a code chunk is command-option-i or control-alt-i.

A code chunk will set off the code and its results in the output document, but you can also print the results of code within a text block by enclosing code like so: ``r code-here``.

6.2.2 Use knitr to Produce a Report

1. Open a new.Rmd script and save it as inflammation_report.Rmd.
2. Copy code from earlier into code chunks to read the inflammation data and plot average inflammation.
3. Add a few notes describing what the code does and what the main findings are. Include an in-line calculation of the median inflammation level.
4. Knit the document and view the HTML result.

6.3 EXERCISE

Follow [Test drive R Markdown by Professor Bryan](#) to create a report (in HTML format) that includes:

- A brief introduction of the bike counts data (where were the data collected, the period, etc.);
- Load the bike counts data in the R code chunks of your R markdown file;
- Embed summary statistics and visualization of the bike counts data in your report;
- [Advanced] Create a vignette for the R package you created for [Create R Package](#) that demonstrates the usage of the package.

6.4 LEARNING MORE

1. [R Markdown from RStudio](#)
2. [R Markdown Cheatsheet](#)
3. [Software Carpentry R Markdown Lesson](#)
4. [Test Drive R Markdown by Professor Bryan](#)
5. [Writing Vignettes with R Markdown](#)

7.0 PART II: OVERVIEW AND INTRODUCTION

Data science is a discipline that combines computing with statistics. A data analysis problem is solved in a series of data-centric steps: data acquisition and representation (Import), data cleaning (Tidy), an iterative sequence of data transformation (Transform), data modeling (Model) and data visualization (Visualize). The end result of the process is to communicate insights obtained from the data (Communicate). Part II of this class will take you through all the steps in the process and will teach you how to approach such problems in a systematic manner. You will learn how to design data analysis pipelines as well as how to implement data analysis pipelines in R. The class will also emphasize how elegant code leads to reproducible science.

7.1 PART II: TIDYVERSE WORKFLOW

1. [Data Importing](#)
2. [Tidy Data](#)
3. [Data Manipulation with dplyr](#)
4. [Data Visualization with ggplot2](#)
5. [Split-Apply-Combine Pattern in Data Modeling](#)
6. [Where To Go From Here](#)

7.2 READINGS

1. R4DS [Chapter 1](#)

7.3 THE TIDYVERSE WORKFLOW

First, you must import your data into R. This typically means that you take data stored in a file, database or web API, and load it into a data frame in R. If you can't get your data into R, you can't do data science on it!

Once you've imported your data, it is a good idea to tidy it. Tidying your data means storing them in a consistent form that matches the semantics of the dataset with the way they are stored. In brief, when your data are tidy, each column is a variable, and each row is an observation. Tidy data are important, because the consistent structure lets you focus your efforts on questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, a common first step is to transform them. Transformation includes narrowing in on observations of interest (like all people in one city, or all data from the last year), creating new variables that are functions of existing variables (like computing velocity from speed and time), and calculating a set of summary statistics (like counts or means). Together, tidying and transforming are called wrangling, because getting your data in a form that's natural to work with often feels like a fight!

Once you have tidy data with the variables you need, there are two main engines of knowledge generation: visualization and modeling. These have complementary strengths and weaknesses so any real analysis will iterate between them many times.

Visualization is a fundamentally human activity. A good visualization will show you things that you did not expect, or raise new questions about the data. A good visualization might also hint that you're asking the wrong question, or you need to collect different data. Visualizations can surprise you, but don't scale particularly well because they require a human to interpret them.

Models are complementary tools for visualization. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, so they generally scale well. Even when they don't, it's usually cheaper to buy more computers than it is to buy more brains! But every model makes assumptions, and by its very nature a model cannot question its own assumptions. That means a model cannot fundamentally surprise you.

The last step of data science is communication, an absolutely critical part of any data analysis project. It doesn't matter how

well your models and visualization have led you to understand the data unless you can also communicate your results to others.

Surrounding all these tools is programming. Programming is a cross-cutting tool that you use in every part of the project. You don't need to be an expert programmer to be a data scientist, but learning more about programming pays off because becoming a better programmer allows you to automate common tasks and solve new problems with greater ease.

You'll use these tools in every data science project, but for most projects they're not enough. There's a rough 80-20 rule at play; you can tackle about 80% of every project using the tools that you'll learn in this book, but you'll need other tools to tackle the remaining 20%. Throughout this book, we'll point you to resources where you can learn more.

7.4 THE TIDYVERSE PACKAGES

- Website: <http://www.tidyverse.org/packages/>
- Install with `install.packages("tidyverse")`

7.5 PIPE OPERATOR

`%>%` pipes an object forward into a function or call expression

- Basic piping
 - `x %>% f` is equivalent to `f(x)`
 - `x %>% f(y)` is equivalent to `f(x, y)`
 - `x %>% f %>% g %>% h` is equivalent to `h(g(f(x)))`
- Using `%>%` with unary function calls

```
require(tidyverse)
iris %>% head
```

```
## 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
iris %>% tail
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 145      6.7      3.3      5.7      2.5 virginica
## 146      6.7      3.0      5.2      2.3 virginica
## 147      6.3      2.5      5.0      1.9 virginica
## 148      6.5      3.0      5.2      2.0 virginica
## 149      6.2      3.4      5.4      2.3 virginica
## 150      5.9      3.0      5.1      1.8 virginica
```

- Placing lhs as the first argument in rhs call

```
## What are the outputs of these two lines?
3 %>% `-(4)
iris %>% head(5)
iris %>% tail(5)
```

- Placing lhs elsewhere in rhs call

```
## What are the outputs of these
3 %>% `-(4, .)
4 %>% c("A", "B", "C", "D", "E")[.]
```

More information available at <http://magrittr.tidyverse.org/>

7.5.1 RStudio Keyboard Shortcut for %>%

- ctrl-Shift-M (Windows)
- Command-Shift-M (Mac)

7.6 CLASS PROJECT

For the class project for Part II, you are expected to follow the tidyverse workflow and use the tidyverse suite of packages to conduct an analytic project using data of your choice. Your project should involve these steps:

1. Data importing and tidying
2. Data transformation
3. Data visualization
4. [Optional] Data modeling
5. [Optional] Reproducible reporting

You can use data from your current work or continue a project you have been working on. Take the feasibility of completing it in a week into consideration when selecting project ideas. If you don't have a feasible project idea at the moment, consider continuing to analyze the bike counts data on Hawthorne Bridge and Tilikum Crossing from Part I. Daily traffic counts data for these two bridges can be found [here](#).

Sample analytic questions are:

- After accounting for seasonal variation, is there a trend in bike traffic across these two bridges?
- How was the bike traffic affected by weather (temperature, precipitation, etc.)?
- Was the bike traffic on the Hawthorne Bridge affected by the opening of Tilikum Crossing?

You can work on any one or all of these questions with visualization and/or modeling after you import, tidy and transform the data when necessary.

8.0 DATA IMPORTING

8.1 READINGS

1. R4DS [Chapter 10](#), [Chapter 11](#)

8.2 TIBBLES VS. DATA.FRAME

8.2.1 Data Frames are Awesome

Whenever you have rectangular, spreadsheet-y data, your default data receptacle in R is a data frame. Do not depart from this without good reason. Data frames are awesome because

- Data frames package related variables neatly together,
 - keeping them in sync vis-a-vis row order and
 - applying any filtering of observations uniformly.
- Most functions for inference, modeling, and graphing are happy to be passed a data frame via a `data =` argument. This has been true in base R for a long time.
- The set of packages known as the [tidyverse](#) takes this one step further and explicitly prioritizes the processing of data frames. This includes popular packages like `dplyr` and `ggplot2`. In fact, the tidyverse prioritizes a special flavor of data frame, called a "tibble."

Data frames — unlike general arrays or, specifically, matrices in R — can hold variables of different flavors, such as character data (subject ID or name), quantitative data (white blood cell count), and categorical information (treated vs. untreated). If you use homogenous structures, like matrices, for data analysis, you are likely to make the terrible mistake of spreading a dataset out over multiple, unlinked objects. Why? Because you can't put character data, such as subject name, into the numeric matrix that holds white blood cell count. This fragmentation is a bad idea.

In [R Coding Basics](#) we use `data.frame` in base R as our main data structure. From now on, we will start to use `tibble` and other functions in [tidyverse](#) as much as possible. This will provide a special type of data frame called a "tibble" that has nice default printing behavior, among other benefits such as speedy performance and better default behavior.

First, install tidyverse packages if you haven't yet; you only need to do this once on your laptop:

```
install.packages("tidyverse")
```

Then load the tidyverse packages:

```
library(tidyverse)
```

There are two main differences in the usage of a tibble vs. a classic `data.frame`: printing and subsetting.

8.2.2 Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on the screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()`:

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
## # A tibble: 1,000 x 5
##           a         b      c      d      e
##       <dtm>    <date> <int>   <dbl> <chr>
## 1 2017-12-19 22:49:16 2018-01-17  1 0.87869674 q
## 2 2017-12-20 11:54:26 2018-01-12  2 0.38758424 i
## 3 2017-12-19 18:21:58 2017-12-25  3 0.68424819 i
## 4 2017-12-20 06:06:34 2017-12-29  4 0.50141580 e
## 5 2017-12-20 09:24:11 2017-12-26  5 0.30633672 r
## 6 2017-12-19 14:19:49 2018-01-11  6 0.12337229 j
## 7 2017-12-19 23:44:36 2017-12-20  7 0.09871534 m
## 8 2017-12-20 09:00:07 2017-12-21  8 0.04929260 m
## 9 2017-12-20 01:21:48 2017-12-20  9 0.91323737 y
## 10 2017-12-19 23:40:43 2018-01-14 10 0.81580178 l
## # ... with 990 more rows
```

Lubridate is an R package for date and time. You need to install the package if you want to replicate the above code on your own computer.

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly `print()` the data frame and control the number of rows (`n`) and the width of the display. `width = Inf` will display all columns:

```
print(mtcars, n = 10, width = Inf)
```

You can also control the default print behavior by setting options:

- `options(tibble.print_max = n)`: (Replace `n` with an actual number, for example, 100) print at most `n` rows. Use `options(dplyr.print_min = Inf)` to always show all rows.
- Use `options(tibble.width = Inf)` to always print all columns, regardless of the width of the screen.

You can see a complete list of options by looking at the package help with `?tibble`.

A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations.

```
View(mtcars)
```

8.2.3 Subsetting

So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, \$ and []. [] can extract by name or column position; \$ only extracts by name but is a little less typing.

```
df <- tibble(  
  x = runif(5),  
  y = rnorm(5)  
)  
  
# Extract by name  
df$x  
## [1] 0.8081816 0.5731858 0.4683709 0.8207141 0.1299996  
df[["x"]]  
## [1] 0.8081816 0.5731858 0.4683709 0.8207141 0.1299996  
# Extract by column position  
df[[1]]  
## [1] 0.8081816 0.5731858 0.4683709 0.8207141 0.1299996
```

Compared to a data.frame, tibbles are more strict: They never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

8.2.4 Converting

Some older functions don't work with tibbles. If you encounter one of these functions, use as.data.frame() to turn a tibble back to a data.frame:

```
class(as.data.frame(df))  
## [1] "data.frame"
```

Or use as_tibble() to convert a data.frame to tibble:

```
class(as_tibble(mtcars))  
## [1] "tbl_df"      "tbl"        "data.frame"
```

The main reason that some older functions don't work with tibble is the [function. We don't use [much because dplyr::filter() and dplyr::select() allow you to solve the same problems with clearer code (but you will learn a little about it in [vector subsetting](#)). With base R data frames, [sometimes returns a data frame, and sometimes returns a vector. With tibbles, [always returns another tibble.

8.3 DATA IMPORT

The most common data in data science are rectangular, spreadsheet-y data that work best to be loaded as an R data frame, and this section focuses on importing data into R as data frames. tidyverse provides the [readr](#) package that reads comma-separated values (csv), tab-delimited values (tsv), and fixed-width files (fwf) as tibbles. Other R packages provide access to Excel spreadsheets, binary statistical data formats (SPSS, SAS, Stata), and relational databases:

- [readr](#) - reads flat files (csv, tsv, fwf) into R
- [readxl](#) - reads Excel files (.xls and .xlsx) into R
- [heaven](#) - reads SPSS, Stata and SAS files into R
- [rstats-db](#) - R interfaces to databases
 - [RMySQL](#)
 - [RSQLite](#)
 - [RPostgres](#)

For this section, we primarily focus on importing data in flat files into data frames. [readr](#) provides these functions:

- [read_csv\(\)](#) reads comma-delimited files, and [read_csv2\(\)](#) reads semicolon separated files (common in countries where , is used as the decimal).
- [read_tsv\(\)](#) reads tab-delimited files, and [read_delim\(\)](#) reads in files with any delimiter.
- [read_fwf\(\)](#) reads fixed-width files. You can specify fields either by their widths with [fwf_widths\(\)](#), or their position with [fwf_positions\(\)](#). [read_table\(\)](#) reads a common variation of fixed width files where columns are separated by white space.
- [read_log\(\)](#) reads Apache-style log files. (But also check out [webreadr](#) which is built on top of [read_log\(\)](#) and provides many more helpful tools.)

These functions all have the similar syntax: Once you've mastered one, you can use the others with ease. For the rest of this chapter, we'll focus on [read_csv\(\)](#). Not only are csv files one of the most common forms of data storage, but once you understand [read_csv\(\)](#), you can easily apply your knowledge to all the other functions in [readr](#).

The first argument to [read_csv\(\)](#) is the most important: It's the path to the file to read.

Download [heights.csv](#) from [this link](#) and save it in the data directory of your RStudio project.

```
heights <- read\_csv("data/heights.csv")
## Parsed with column specification:
## cols(
##   earn = col_double(),
##   height = col_double(),
##   sex = col_character(),
##   ed = col_integer(),
##   age = col_integer(),
##   race = col_character()
## )
heights
```

```
## # A tibble: 1,192 x 6
##   earn height sex   ed age  race
##   <dbl>   <dbl> <chr> <int> <int> <chr>
## 1 50000 74.42444 male   16  45  white
## 2 60000 65.53754 female 16  58  white
## 3 30000 63.62920 female 16  29  white
## 4 50000 63.10856 female 16  91  other
## 5 51000 63.40248 female 17  39  white
## 6 9000 64.39951 female 15  26  white
## 7 29000 61.65633 female 12  49  white
## 8 32000 72.69854 male   17  46  white
## 9 2000 72.03947 male   15  21  hispanic
## 10 27000 72.23493 male   12  26  white
## # ... with 1,182 more rows
```

When you run `read_csv()`, it prints out a column specification that gives the name and type of each column. That's an important part of the `readr` package and more details of how it works are available in the [parsing a file section of R4DS](#).

`read_csv()` uses the first line of the data for the column names, which is a very common convention. There are two cases where you might want to tweak this behavior:

1. Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first `n` lines, or use `comment = "#"` to drop all lines that start with (e.g.) `#`.

```
read_csv("The first line of metadata
The second line of metadata
x,y,z
1,2,3", skip = 2)
## # A tibble: 1 x 3
##   x     y     z
##   <int> <int> <int>
## 1     1     2     3
read_csv("# A comment I want to skip
x,y,z
1,2,3", comment = "#")
## # A tibble: 1 x 3
##   x     y     z
##   <int> <int> <int>
## 1     1     2     3
```

2. The data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings, and instead label them sequentially from `X1` to `Xn`:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
## # A tibble: 2 x 3
##   X1    X2    X3
## 1     1     2     3
## 2     4     5     6
```

```
## <int> <int> <int>
## 1    1    2    3
## 2    4    5    6
```

("\\n" is a convenient shortcut for adding a new line. You'll learn more about it and other types of string escape in [string basics](#).)

Alternatively, you can pass `col_names` a character vector to use as the column names:

```
read_csv("1,2,3\\n4,5,6", col_names = c("x", "y", "z"))
## # A tibble: 2 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

Another option that commonly needs tweaking is `na`: This specifies the value (or values) that are used to represent missing values in your file:

```
read_csv("a,b,c\\n1,2,.", na = ".")
## # A tibble: 1 x 3
##       a     b     c
##   <int> <int> <chr>
## 1     1     2 <NA>
```

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab-separated files with `read_tsv()` and fixed-width files with `read_fwf()`. To read more challenging files, you'll need to learn more about how `readr` parses each column, turning them into R vectors.

8.4 EXERCISE

[Link to the README file for the data](#)

1. What is the difference between a `data.frame` and `tibble`? How do you convert between them?
2. Import the bike counts data for [Hawthorne](#) and [Tilikum](#) in Microsoft Excel format;
3. Import the [Portland weather data in csv format](#);
4. [Challenge] Import the [Portland weather data in fixed-width format](#);
5. For those already familiar with R, create an R script that loads, cleans, and visualizes the bike counts data as well as temperature and precipitation data (using data from Weather Station USC00356750).

For those not yet familiar with R, think about how you would go about doing these tasks with the software you are most comfortable with.

8.5 LEARNING MORE

- [R Data Import Tutorial](#) by Data Camp
- [Importing Data](#) and [Exporting Data](#) by R Programming @ UC.
- [Data Import & Export in R](#) by National Park Services
- [Scrape web data with APIs](#)

9.0 TIDY DATA

9.1 READINGS

1. [R4DS Chapter 12](#)
2. [Wickham, Hadley, 2014. Tidy Data, Journal of Statistical Software Vol 59 \(2014\), Issue 10, 10.18637/jss.v059.i10](#)

9.2 TIDY DATA

There are three interrelated rules, which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

That interrelationship leads to an even simpler set of practical instructions:

- Put each dataset in a tibble.
- Put each variable in a column.

In this session, you will learn a consistent way to organize your data in R, an organization called **tidy data**. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand. This session will give you a practical introduction to tidy data and the accompanying tools in the **tidyr** package.

9.3 LESSON

table1 - table4 (loaded with the tidyverse package) below shows the same data organized in four different ways. Each dataset shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organizes the values in a different way:

```
library(tidyverse)
```

```
table1
## # A tibble: 6 x 4
##   country year cases population
##   <chr> <int> <int>    <int>
## 1 Afghanistan 1999   745  19987071
## 2 Afghanistan 2000  2666 20595360
## 3   Brazil 1999 37737 172006362
## 4   Brazil 2000  80488 174504898
## 5    China 19992122581272915272
## 6    China 20002137661280428583
```


table2

A tibble: 12 x 4

	country	year	type	count
##	<chr>	<int>	<chr>	<int>
## 1	Afghanistan	1999	cases	745
## 2	Afghanistan	1999	population	19987071
## 3	Afghanistan	2000	cases	2666
## 4	Afghanistan	2000	population	20595360
## 5	Brazil	1999	cases	37737
## 6	Brazil	1999	population	172006362
## 7	Brazil	2000	cases	80488
## 8	Brazil	2000	population	174504898
## 9	China	1999	cases	212258
## 10	China	1999	population	1272915272
## 11	China	2000	cases	213766
## 12	China	2000	population	1280428583

table3

A tibble: 6 x 3

##	country	year	rate
## *	<chr>	<int>	<chr>
## 1	Afghanistan	1999	745/19987071
## 2	Afghanistan	2000	2666/20595360
## 3	Brazil	1999	37737/172006362
## 4	Brazil	2000	80488/174504898
## 5	China	1999	212258/1272915272
## 6	China	2000	213766/1280428583

Spread across two tibbles

table4a *# cases*

A tibble: 3 x 3

##	country	`1999`	`2000`
## *	<chr>	<int>	<int>
## 1	Afghanistan	745	2666
## 2	Brazil	37737	80488
## 3	China	212258	213766

table4b *# population*

A tibble: 3 x 3

##	country	`1999`	`2000`
## *	<chr>	<int>	<int>
## 1	Afghanistan	19987071	20595360
## 2	Brazil	172006362	174504898
## 3	China	1272915272	1280428583

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

9.3.1 Spreading and Gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy.

There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data are often organized to facilitate some use other than analysis. For example, data are often organized to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

Typically, a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidy: `gather()` and `spread()`.

9.3.1.1 Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take table4a: the column names 1999 and 2000 represent values of the year variable, and each row represents two observations, not one.

```
table4a
## # A tibble: 3 x 3
##   country `1999` `2000`
## *   <chr> <int> <int>
## 1 Afghanistan  745  2666
## 2   Brazil 37737 80488
## 3    China 212258 213766
```

To tidy a dataset like this, we need to **gather** those columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns that represent values, not variables. In this example, those are the columns 1999 and 2000.
- The name of the variable whose values form the column names. I call that the key, and here it is year.
- The name of the variable whose values are spread over the cells. I call that value, and here it's the number of cases.

Together those parameters generate the call to `gather()`:

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
## # A tibble: 6 x 3
##   country year cases
##   <chr> <chr> <int>
## 1 Afghanistan 1999 745
## 2   Brazil 1999 37737
## 3    China 1999 212258
## 4 Afghanistan 2000 2666
## 5   Brazil 2000 80488
## 6    China 2000 213766
```

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. Note that "1999" and "2000" are non-syntactic names so we have to surround them in backticks. We will learn more about `select` in [Data manipulation with dplyr](#).

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

Figure 9.1 Gathering table4 into a tidy form

In the final result the gathered columns are dropped, and we get new key and value columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure 9.1. We can use `gather()` to tidy table4b in a similar fashion. The only difference is the variable stored in the cell values:

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
## # A tibble: 6 x 3
##   country year population
##   <chr> <chr> <int>
## 1 Afghanistan 1999 19987071
## 2   Brazil 1999 172006362
## 3    China 1999 1272915272
## 4 Afghanistan 2000 20595360
## 5   Brazil 2000 174504898
## 6    China 2000 1280428583
```

To combine the tidied versions of table4a and table4b into a single tibble, we need to use `dplyr::left_join()`, which you'll learn about in [Working with two datasets](#).

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
## # A tibble: 6 x 4
##   country year cases population
##   <chr> <chr> <int>    <int>
## 1 Afghanistan 1999    745  19987071
## 2   Brazil 1999  37737 172006362
## 3    China 1999 212258 1272915272
## 4 Afghanistan 2000   2666  20595360
## 5   Brazil 2000  80488 174504898
## 6    China 2000 213766 1280428583
```

9.3.1.2 Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take table2: an observation is a country in a year, but each observation is spread across two rows.

```
table2
## # A tibble: 12 x 4
##   country year   type    count
##   <chr> <int>  <chr>    <int>
## 1 Afghanistan 1999   cases     745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000   cases     2666
## 4 Afghanistan 2000 population 20595360
## 5   Brazil 1999   cases     37737
## 6   Brazil 1999 population 172006362
## 7   Brazil 2000   cases     80488
## 8   Brazil 2000 population 174504898
## 9    China 1999   cases     212258
## 10   China 1999 population 1272915272
## 11   China 2000   cases     213766
## 12   China 2000 population 1280428583
```

To tidy this up, we first analyze the representation in a similar way to `gather()`. This time, however, we only need two parameters:

- The column that contains variable names, the key column. Here, it's `type`.
- The column that contains values forms multiple variables, the value column. Here it's `count`.

Once we've figured that out, we can use `spread()`, as shown programmatically below, and visually in Figure 9.2.

```
spread(table2, key = type, value = count)
## # A tibble: 6 x 4
##   country year cases population
## *   <chr> <int> <int>   <int>
## 1 Afghanistan 1999   745 19987071
## 2 Afghanistan 2000  2666 20595360
## 3   Brazil 1999 37737 172006362
## 4   Brazil 2000 80488 174504898
## 5    China 1999 212258 1272915272
## 6    China 2000 213766 1280428583
```

Figure 9.2 Spreading table2 makes it tidy

As you might have guessed from the common key and value arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

9.3.2 Separating and Uniting

So far, you've learned how to tidy table2 and table4, but not table3. table3 has a different problem: we have one column (rate) that contains two variables (cases and population). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

9.3.2.1 *Separate*

`separate()` pulls one column apart into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
## # A tibble: 6 x 3
##   country year      rate
## *   <chr> <int>    <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3   Brazil 1999 37737/172006362
## 4   Brazil 2000 80488/174504898
## 5    China 1999 212258/1272915272
## 6    China 2000 213766/1280428583
```

The `rate` column contains both cases and population variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown in Figure 9.3 and the code below.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
## # A tibble: 6 x 4
##   country year cases population
## *   <chr> <int> <chr>    <chr>
## 1 Afghanistan 1999  745  19987071
## 2 Afghanistan 2000 2666  20595360
## 3   Brazil 1999 37737 172006362
## 4   Brazil 2000 80488 174504898
## 5    China 1999 212258 1272915272
## 6    China 2000 213766 1280428583
```

Figure 9.3 Separating table3 makes it tidy

By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e., a character that isn't a number or letter). For example, in the code above, `separate()` split the values of `rate` at the slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the code above as:

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

(Formally, `sep` is a regular expression, which you can learn more about in the [Strings chapter of R4DS](#).)

Look carefully at the column types: you'll notice that `cases` and `population` are character columns. This is the default behavior in `separate()`: It leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try converting to better types using `convert = TRUE`:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
## # A tibble: 6 x 4
##   country year cases population
## *   <chr> <int> <int>    <int>
## 1 Afghanistan 1999   745 19987071
## 2 Afghanistan 2000  2666 20595360
## 3   Brazil 1999 37737 172006362
## 4   Brazil 2000 80488 174504898
```

```
## 5    China 1999 212258 1272915272
## 6    China 2000 213766 1280428583
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far left of the strings; negative values start at -1 on the far right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This makes these data less tidy, but is useful in other cases, as you'll see in a little bit.

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
## # A tibble: 6 x 4
##   country century year      rate
## *   <chr>   <chr> <chr>    <chr>
## 1 Afghanistan    19  99  745/19987071
## 2 Afghanistan    20  00 2666/20595360
## 3   Brazil      19  99 37737/172006362
## 4   Brazil      20  00 80488/174504898
## 5    China      19  99 212258/1272915272
## 6    China      20  00 213766/1280428583
```

9.3.2.2 *Unite*

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

Figure 9.4 Uniting table5 makes it tidy

We can use `unite()` to rejoin the *century* and *year* columns that we created in the last example (Figure 9.4). The data are saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` style:

```
table5 %>%
  unite(new, century, year)
## # A tibble: 6 x 3
##   country new      rate
## *   <chr> <chr>    <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3   Brazil 19_99 37737/172006362
## 4   Brazil 20_00 80488/174504898
## 5    China 19_99 212258/1272915272
## 6    China 20_00 213766/1280428583
```

In this case, we also need to use the `sep` argument. The default will place an underscore (`_`) between the values from different columns. Here we don't want any separator so we use `"`:

```
table5 %>%
  unite(new, century, year, sep = "")
## # A tibble: 6 x 3
##   country new      rate
## *   <chr> <chr>    <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3   Brazil 1999 37737/172006362
## 4   Brazil 2000 80488/174504898
## 5    China 1999 212258/1272915272
## 6    China 2000 213766/1280428583
```

9.4 EXERCISE

1. Are the bike counts data for the two bridges tidy data?
2. If not, why not? And how can we tidy it?
3. After tidying the bike counts, using functions in the `tidyr` package, create tables summarizing the average bike counts by bridge and day of week in two different formats:

Table 9.1 Bike counts by day of week and bridge (1st format)

Bridge	Sun	Mon	Tue	Wed	Thur	Fri	Sat
Hawthorne							
Tilikum							

Table 9.2 Bike counts by day of week and bridge (2nd format)

Day of Week	Hawthorne	Tilikum
Fri		
Mon		

Sat
Sun
Thur
Tue
Wed

9.5 LEARNING MORE

1. [Dataframe Manipulation with tidyr](#)
2. [Data Wrangling Cheatsheet](#)
3. [Introduction to tidyr](#)

10.0 DATA MANIPULATION WITH DPLYR

10.1 READINGS

1. [R4DS Data Transformation](#)
2. [R4DS Relational Data](#)

10.2 OVERVIEW

dplyr is a grammar of data manipulation, providing a consistent set of verbs that helps you solve the most common data manipulation challenges:

- `mutate()` adds new variables that are functions of existing variables.
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

These all combine naturally with `group_by()` which allows you to perform any operation "by group." You can learn more about them in `vignette("dplyr")`. As well as these single-table verbs, dplyr also provides a variety of two-table verbs, which you can learn about in `vignette("two-table")`.

dplyr is designed to abstract over how the data are stored. That means as well as working with local data frames, you can also work with remote database tables using exactly the same R code. Install the `dbplyr` package and then read `vignette("databases", package = "dbplyr")`.

10.3 LESSON

The lesson below is adapted from [UBC's Stat 545 course materials](#) by Professor Jenny Bryan.

10.3.1 Prerequisites

Load dplyr (or tidyverse, which will load dplyr) and gapminder (install it if not yet installed):

```
library(gapminder)
library(tidyverse)
```

10.3.2 Use `filter()` to Subset Data Row-wise

`filter()` takes logical expressions and returns the rows for which all are TRUE.

```
filter(gapminder, lifeExp < 29)
## # A tibble: 2 x 6
##   country continent year lifeExp  pop gdpPercap
##   <fctr>   <fctr> <int>  <dbl> <int>   <dbl>
```

```
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453
## 2 Rwanda Africa 1992 23.599 7290203 737.0686
```

```
filter(gapminder, country == "Rwanda", year > 1979)
```

```
## # A tibble: 6 x 6
```

```
## country continent year lifeExp pop gdpPercap
## <fctr> <fctr> <int> <dbl> <int> <dbl>
## 1 Rwanda Africa 1982 46.218 5507565 881.5706
## 2 Rwanda Africa 1987 44.020 6349365 847.9912
## 3 Rwanda Africa 1992 23.599 7290203 737.0686
## 4 Rwanda Africa 1997 36.087 7212583 589.9445
## 5 Rwanda Africa 2002 43.413 7852401 785.6538
## 6 Rwanda Africa 2007 46.242 8860588 863.0885
```

```
filter(gapminder, country %in% c("Rwanda", "Afghanistan"))
```

```
## # A tibble: 24 x 6
```

```
## country continent year lifeExp pop gdpPercap
## <fctr> <fctr> <int> <dbl> <int> <dbl>
## 1 Afghanistan Asia 1952 28.801 8425333 779.4453
## 2 Afghanistan Asia 1957 30.332 9240934 820.8530
## 3 Afghanistan Asia 1962 31.997 10267083 853.1007
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414
## # ... with 14 more rows
```

Under no circumstances should you subset your data the way I did at first:

```
excerpt <- gapminder[241:252, ]
```

Why is this a terrible idea?

- It is not self-documenting. What is so special about rows 241 through 252?
- It is fragile. This line of code will produce different results if someone changes the row order of gapminder (e.g., sorts the data earlier in the script).

```
filter(gapminder, country == "Canada")
```

This call explains itself and is fairly robust.

10.3.3 Use select() to Subset the Data on Variables or Columns

Use select() to subset the data on variables or columns. Here's a conventional call:

```
select(gapminder, year, lifeExp)
## # A tibble: 1,704 x 2
##   year lifeExp
##   <int> <dbl>
## 1 1952 28.801
## 2 1957 30.332
## 3 1962 31.997
## 4 1967 34.020
## 5 1972 36.088
## 6 1977 38.438
## 7 1982 39.854
## 8 1987 40.822
## 9 1992 41.674
## 10 1997 41.763
## # ... with 1,694 more rows
```

Think: "Take gapminder, then select the variables year and lifeExp, then show the first four rows."

10.3.4 Revel in the Convenience

Here are the data for Cambodia, taking certain variables:

```
gapminder %>%
  filter(country == "Cambodia") %>%
  select(year, lifeExp)
## # A tibble: 12 x 2
##   year lifeExp
##   <int> <dbl>
## 1 1952 39.417
## 2 1957 41.366
## 3 1962 43.415
## 4 1967 45.415
## 5 1972 40.317
## 6 1977 31.220
## 7 1982 50.957
## 8 1987 53.914
## 9 1992 55.803
## 10 1997 56.534
## 11 2002 56.752
## 12 2007 59.723
```

10.3.5 Pure, Predictable, Pipeable

We've barely scratched the surface of dplyr but I want to point out key principles you may start to appreciate. If you're new to R or "programming with data," feel free skip this section and [move on](#).

dplyr's verbs, such as `filter()` and `select()`, are what's called [pure functions](#). To quote from Wickham's [Advanced R Programming book](#):

The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no side effects: they don't affect the state of the world in any way apart from the value they return.

In fact, these verbs are a special case of pure functions: they take the same flavor of the object as input and output. Namely, a data frame or one of the other data receptacles dplyr supports.

And finally, the data are **always** the very first argument of the verb functions.

This set of deliberate design choices, together with the new pipe operator, produces a highly effective, low friction [domain-specific language](#) for data analysis.

10.3.6 Use `mutate()` to Add New Variables

Imagine we wanted to recover each country's GDP. After all, the Gapminder data have a variable for population and GDP per capita. Let's multiply them together.

`mutate()` is a function that defines and inserts new variables into a tibble. You can refer to existing variables by name.

```
gapminder %>%  
  mutate(gdp = pop * gdpPercap)  
## # A tibble: 1,704 x 7  
##   country continent year lifeExp  pop gdpPercap  gdp  
##   <fctr>   <fctr> <int>  <dbl> <int>   <dbl>   <dbl>  
## 1 Afghanistan Asia  1952  28.801 8425333 779.4453 6567086330  
## 2 Afghanistan Asia  1957  30.332 9240934 820.8530 7585448670  
## 3 Afghanistan Asia  1962  31.997 10267083 853.1007 8758855797  
## 4 Afghanistan Asia  1967  34.020 11537966 836.1971 9648014150  
## 5 Afghanistan Asia  1972  36.088 13079460 739.9811 9678553274  
## 6 Afghanistan Asia  1977  38.438 14880372 786.1134 11697659231  
## 7 Afghanistan Asia  1982  39.854 12881816 978.0114 12598563401  
## 8 Afghanistan Asia  1987  40.822 13867957 852.3959 11820990309  
## 9 Afghanistan Asia  1992  41.674 16317921 649.3414 10595901589  
## 10 Afghanistan Asia  1997  41.763 22227415 635.3414 14121995875  
## # ... with 1,694 more rows
```

Hmm ... those GDP numbers are almost uselessly large and abstract. Consider the [advice of Randall Munroe of xkcd](#):

One thing that bothers me is large numbers presented without context... 'If I added a zero to this number, would the sentence containing it mean something different to me?' If the answer is 'no,' maybe the number has no business being in the sentence in the first place."

Maybe it would be more meaningful to consumers of my tables and figures to stick with GDP per capita. But what if I reported GDP per capita, relative to a benchmark country. I need to create a new variable that is gdpPercap divided by Canadian gdpPercap, taking care that I always divide two numbers that pertain to the same year.

How I achieve this:

- Filter down to the rows for Canada
- Create a new temporary variable in gapminder:
 - Extract the gdpPercap variable from the Canadian data.
 - Replicate it once per country in the dataset, so it has the right length.
- Divide raw gdpPercap by this Canadian figure.
- Discard the temporary variable of replicated Canadian gdpPercap.

```
ctib <- gapminder %>%
  filter(country == "Canada") %>%
  select(year, BasegdpPercap=gdpPercap)
```

```
ctib
## # A tibble: 12 x 2
##   year BasegdpPercap
##   <int>     <dbl>
## 1 1952    11367.16
## 2 1957    12489.95
## 3 1962    13462.49
## 4 1967    16076.59
## 5 1972    18970.57
## 6 1977    22090.88
## 7 1982    22898.79
## 8 1987    26626.52
## 9 1992    26342.88
## 10 1997   28954.93
## 11 2002   33328.97
## 12 2007   36319.24
```

```
my_gap <- gapminder %>%
  inner_join(ctib, by="year") %>%
  mutate(gdpPercapRel = gdpPercap / BasegdpPercap,
         BasegdpPercap = NULL)
```

```
my_gap
## # A tibble: 1,704 x 7
##   country continent year lifeExp   pop gdpPercap gdpPercapRel
##   <fctr>   <fctr> <int>  <dbl> <int>   <dbl>      <dbl>
## 1 Afghanistan Asia  1952  28.801 8425333 779.4453 0.06856992
## 2 Afghanistan Asia  1957  30.332 9240934 820.8530 0.06572108
## 3 Afghanistan Asia  1962  31.997 10267083 853.1007 0.06336874
```

```
## 4 Afghanistan Asia 1967 34.020 11537966 836.1971 0.05201335
## 5 Afghanistan Asia 1972 36.088 13079460 739.9811 0.03900679
## 6 Afghanistan Asia 1977 38.438 14880372 786.1134 0.03558542
## 7 Afghanistan Asia 1982 39.854 12881816 978.0114 0.04271018
## 8 Afghanistan Asia 1987 40.822 13867957 852.3959 0.03201305
## 9 Afghanistan Asia 1992 41.674 16317921 649.3414 0.02464959
## 10 Afghanistan Asia 1997 41.763 22227415 635.3414 0.02194243
## # ... with 1,694 more rows
```

Note that, `mutate()` builds new variables sequentially so you can reference earlier ones (like `tmp`) when defining later ones (like `gdpPercapRel`). Also, you can get rid of a variable by setting it to `NULL`.

How could we sanity check that this worked? The Canadian values for `gdpPercapRel` better all be 1!

```
my_gap %>%
  filter(country == "Canada") %>%
  select(country, year, gdpPercapRel)
## # A tibble: 12 x 3
##   country year gdpPercapRel
##   <fctr> <int>     <dbl>
## 1 Canada 1952         1
## 2 Canada 1957         1
## 3 Canada 1962         1
## 4 Canada 1967         1
## 5 Canada 1972         1
## 6 Canada 1977         1
## 7 Canada 1982         1
## 8 Canada 1987         1
## 9 Canada 1992         1
## 10 Canada 1997        1
## 11 Canada 2002        1
## 12 Canada 2007        1
```

I perceive Canada to be a "high GDP" country, so I predict that the distribution of `gdpPercapRel` is located below 1, possibly even well below. Check your intuition!

```
summary(my_gap$gdpPercapRel)
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.007236 0.061648 0.171521 0.326659 0.446564 9.534690
```

The relative GDP per capita numbers are, in general, well below 1. We see that most of the countries covered by this dataset have substantially lower GDP per capita, relative to Canada, across the entire time period.

Remember: Trust No One. Including (especially?) yourself. Always try to find a way to check that you've done what you meant to. Prepare to be horrified.

10.3.7 Use `arrange()` to Row-order Data in a Principled Way

`arrange()` reorders the rows in a data frame. Imagine you wanted these data ordered by year then country, as opposed to by country then year.

```
my_gap %>%  
  arrange(year, country)  
## # A tibble: 1,704 x 7  
##   country continent year lifeExp   pop gdpPercap gdpPercapRel  
##   <fctr>   <fctr> <int>  <dbl> <int>   <dbl>     <dbl>  
## 1 Afghanistan Asia  1952  28.801 8425333  779.4453  0.06856992  
## 2 Albania Europe  1952  55.230 1282697 1601.0561  0.14084925  
## 3 Algeria Africa  1952  43.077 9279525 2449.0082  0.21544589  
## 4 Angola Africa  1952  30.015 4232095 3520.6103  0.30971764  
## 5 Argentina Americas 1952  62.485 17876956 5911.3151  0.52003442  
## 6 Australia Oceania 1952  69.120 8691212 10039.5956  0.88321046  
## 7 Austria Europe  1952  66.800 6927772 6137.0765  0.53989527  
## 8 Bahrain Asia  1952  50.939 120447 9867.0848  0.86803421  
## 9 Bangladesh Asia  1952  37.484 46886859 684.2442  0.06019482  
## 10 Belgium Europe 1952  68.000 8730405 8343.1051  0.73396559  
## # ... with 1,694 more rows
```

Or maybe you want just the data from 2007, sorted on life expectancy?

```
my_gap %>%  
  filter(year == 2007) %>%  
  arrange(lifeExp)  
## # A tibble: 142 x 7  
##   country continent year lifeExp   pop gdpPercap  
##   <fctr>   <fctr> <int>  <dbl> <int>   <dbl>  
## 1 Swaziland Africa  2007  39.613 1133066 4513.4806  
## 2 Mozambique Africa  2007  42.082 19951656 823.6856  
## 3 Zambia Africa  2007  42.384 11746035 1271.2116  
## 4 Sierra Leone Africa  2007  42.568 6144562 862.5408  
## 5 Lesotho Africa  2007  42.592 2012649 1569.3314  
## 6 Angola Africa  2007  42.731 12420476 4797.2313  
## 7 Zimbabwe Africa  2007  43.487 12311143 469.7093  
## 8 Afghanistan Asia  2007  43.828 31889923 974.5803  
## 9 Central African Republic Africa 2007  44.741 4369038 706.0165  
## 10 Liberia Africa  2007  45.678 3193942 414.5073  
## # ... with 132 more rows, and 1 more variables: gdpPercapRel <dbl>
```

Oh, you'd like to sort life expectancy in **descending** order? Then use `desc()`.

```
my_gap %>%
  filter(year == 2007) %>%
  arrange(desc(lifeExp))
## # A tibble: 142 x 7
##   country continent year lifeExp    pop gdpPercap
##   <fctr>    <fctr> <int>  <dbl>  <int>    <dbl>
## 1      Japan    Asia  2007  82.603 127467972 31656.07
## 2 Hong Kong, China Asia  2007  82.208  6980412 39724.98
## 3      Iceland Europe  2007  81.757   301931 36180.79
## 4 Switzerland Europe  2007  81.701  7554661 37506.42
## 5    Australia Oceania  2007  81.235 20434176 34435.37
## 6      Spain Europe  2007  80.941 40448191 28821.06
## 7      Sweden Europe  2007  80.884  9031088 33859.75
## 8      Israel Asia   2007  80.745  6426679 25523.28
## 9      France Europe  2007  80.657 61083916 30470.02
## 10     Canada Americas 2007  80.653 33390141 36319.24
## # ... with 132 more rows, and 1 more variables: gdpPercapRel <dbl>
```

I advise that your analyses NEVER rely on rows or variables being in a specific order. But it's still true that human beings write the code and the interactive development process can be much nicer if you reorder the rows of your data as you go along. Also, once you are preparing tables for human eyeballs, it is imperative that you step up and take control of row order.

10.3.8 Use `rename()` to Rename Variables

The Gapminder uses `camelCase` for variable naming; an alternative convention is `snake_case`. Let's rename some variables to use `snake_case`!

```
my_gap %>%
  rename(life_exp = lifeExp,
         gdp_percap = gdpPercap,
         gdp_percap_rel = gdpPercapRel)
## # A tibble: 1,704 x 7
##   country continent year life_exp    pop gdp_percap gdp_percap_rel
##   <fctr>    <fctr> <int>  <dbl>  <int>    <dbl>    <dbl>
## 1 Afghanistan Asia  1952  28.801  8425333  779.4453  0.06856992
## 2 Afghanistan Asia  1957  30.332  9240934  820.8530  0.06572108
## 3 Afghanistan Asia  1962  31.997 10267083  853.1007  0.06336874
## 4 Afghanistan Asia  1967  34.020 11537966  836.1971  0.05201335
## 5 Afghanistan Asia  1972  36.088 13079460  739.9811  0.03900679
## 6 Afghanistan Asia  1977  38.438 14880372  786.1134  0.03558542
## 7 Afghanistan Asia  1982  39.854 12881816  978.0114  0.04271018
## 8 Afghanistan Asia  1987  40.822 13867957  852.3959  0.03201305
## 9 Afghanistan Asia  1992  41.674 16317921  649.3414  0.02464959
## 10 Afghanistan Asia  1997  41.763 22227415  635.3414  0.02194243
## # ... with 1,694 more rows
```

I did NOT assign the post-rename object back to `my_gap` because that would make the chunks in this tutorial harder to copy/paste and run out of order. In real life, I would probably assign this back to `my_gap`, in a data preparation script, and proceed with the new variable names.

10.3.9 `select()` Can Rename and Reposition Variables

You've seen simple uses of `select()`. There are two tricks you might enjoy:

1. `select()` can rename the variables you request to keep.
2. `select()` can be used with `everything()` to hoist a variable up to the front of the tibble.

```
my_gap %>%  
  filter(country == "Burundi", year > 1996) %>%  
  select(yr = year, lifeExp, gdpPercap) %>%  
  select(gdpPercap, everything())  
## # A tibble: 3 x 3  
##   gdpPercap  yr lifeExp  
##   <dbl> <int> <dbl>  
## 1 463.1151 1997 45.326  
## 2 446.4035 2002 47.360  
## 3 430.0707 2007 49.580
```

`everything()` is one of several helpers for variable selection. Read its help feature to see the rest.

10.3.10 `group_by()` is a Mighty Weapon

I have found that collaborators love to ask seemingly innocuous questions like, "Which country experienced the sharpest five-year drop in life expectancy?" In fact, that is a totally natural question to ask. But if you are using a language that doesn't know about data, it's an incredibly annoying question to answer.

`dplyr` offers powerful tools to solve this class of problem.

- `group_by()` adds extra structure to your dataset — grouping information — which lays the groundwork for computations within the groups.
- `summarize()` takes a dataset with n observations, computes requested summaries, and returns a dataset with one observation.
- Window functions take a dataset with n observations and returns a dataset with n observations.
- `mutate()` and `summarize()` will honor groups.
- You can also do very general computations on your groups with `do()`, though elsewhere in this course I advocate for other approaches that I find more intuitive, using the `purrr` package.

Combined with the verbs you already know, these new tools allow you to solve an extremely diverse set of problems with relative ease.

10.3.10.1 Counting Things Up

Let's start with simple counting. How many observations do we have per continent?

```
my_gap %>%  
  group_by(continent) %>%  
  summarize(n = n())  
## # A tibble: 5 x 2  
##   continent     n  
##   <fctr> <int>  
## 1 Africa   624  
## 2 Americas 300  
## 3 Asia    396  
## 4 Europe   360  
## 5 Oceania   24
```

Let us pause here to think about the tidyverse. You could get these same frequencies using `table()` from base R.

```
table(gapminder$continent)  
##  
## Africa Americas Asia Europe Oceania  
## 624 300 396 360 24  
  
str(table(gapminder$continent))  
## 'table' int [1:5(1d)] 624 300 396 360 24  
## - attr(*, "dimnames")=List of 1  
## ..$ : chr [1:5] "Africa" "Americas" "Asia" "Europe" ...
```

But the object of the class `table` that is returned makes downstream computation a bit fiddlier than you'd like. For example, it's too bad the continent levels come back only as *names* and not as a proper factor, with the original set of levels. This is an example of how the tidyverse smooths transitions where you want the output of step *i* to become the input of step *i* + 1.

The `tally()` function is a convenient function that knows to count rows. It honors groups.

```
my_gap %>%  
  group_by(continent) %>%  
  tally()  
## # A tibble: 5 x 2  
##   continent     n  
##   <fctr> <int>  
## 1 Africa   624  
## 2 Americas 300  
## 3 Asia    396  
## 4 Europe   360  
## 5 Oceania   24
```

The `count()` function is an even more convenient function that does both grouping and counting.

```
my_gap %>%  
  count(continent)  
## # A tibble: 5 x 2  
##   continent     n  
##   <fctr> <int>  
## 1 Africa  624  
## 2 Americas 300  
## 3 Asia    396  
## 4 Europe  360  
## 5 Oceania  24
```

What if we wanted to add the number of unique countries for each continent? You can compute multiple summaries inside `summarize()`. Use the `n_distinct()` function to count the number of distinct countries within each continent.

```
my_gap %>%  
  group_by(continent) %>%  
  summarize(n = n(),  
            n_countries = n_distinct(country))  
## # A tibble: 5 x 3  
##   continent     n n_countries  
##   <fctr> <int>   <int>  
## 1 Africa  624      52  
## 2 Americas 300      25  
## 3 Asia    396      33  
## 4 Europe  360      30  
## 5 Oceania  24       2
```

10.3.10.2 General Summarization

The functions you'll apply within `summarize()` include classical statistical summaries, like `mean()`, `median()`, `var()`, `sd()`, `mad()`, `IQR()`, `min()`, and `max()`. Remember they are functions that take n inputs and distill them down into one output.

Although this may be statistically ill-advised, let's compute the average life expectancy by continent.

```
my_gap %>%  
  group_by(continent) %>%  
  summarize(avg_lifeExp = mean(lifeExp))  
## # A tibble: 5 x 2  
##   continent avg_lifeExp  
##   <fctr>      <dbl>  
## 1 Africa  48.86533  
## 2 Americas 64.65874
```

```
## 3 Asia 60.06490
## 4 Europe 71.90369
## 5 Oceania 74.32621
```

summarize_each() applies the same summary function(s) to multiple variables. Let's compute average and median life expectancy and GDP per capita by continent by year ... but only for 1952 and 2007.

```
my_gap %>%
  filter(year %in% c(1952, 2007)) %>%
  group_by(continent, year) %>%
  summarise_each(funs(mean, median), lifeExp, gdpPercap)
## # A tibble: 10 x 6
## # Groups:   continent [?]
##   continent year lifeExp_mean gdpPercap_mean lifeExp_median
##   <fctr> <int>      <dbl>      <dbl>      <dbl>
## 1 Africa 1952    39.13550    1252.572    38.8330
## 2 Africa 2007    54.80604    3089.033    52.9265
## 3 Americas 1952    53.27984    4079.063    54.7450
## 4 Americas 2007    73.60812    11003.032    72.8990
## 5 Asia 1952    46.31439    5195.484    44.8690
## 6 Asia 2007    70.72848    12473.027    72.3960
## 7 Europe 1952    64.40850    5661.057    65.9000
## 8 Europe 2007    77.64860    25054.482    78.6085
## 9 Oceania 1952    69.25500    10298.086    69.2550
## 10 Oceania 2007    80.71950    29810.188    80.7195
## # ... with 1 more variables: gdpPercap_median <dbl>
```

Let's focus just on Asia. What are the minimum and maximum life expectancies seen by year?

```
my_gap %>%
  filter(continent == "Asia") %>%
  group_by(year) %>%
  summarize(min_lifeExp = min(lifeExp), max_lifeExp = max(lifeExp))
## # A tibble: 12 x 3
##   year min_lifeExp max_lifeExp
##   <int>      <dbl>      <dbl>
## 1 1952    28.801    65.390
## 2 1957    30.332    67.840
## 3 1962    31.997    69.390
## 4 1967    34.020    71.430
## 5 1972    36.088    73.420
## 6 1977    31.220    75.380
## 7 1982    39.854    77.110
## 8 1987    40.822    78.670
## 9 1992    41.674    79.360
## 10 1997    41.763    80.690
```

```
## 11 2002    42.129    82.000
## 12 2007    43.828    82.603
```

Of course, it would be much more interesting to see *which* country contributed these extreme observations. Is the minimum (maximum) always coming from the same country? We tackle that with window functions shortly.

10.3.11 Grouped Mutate

Sometimes you don't want to collapse the n rows for each group into one row. You want to keep your groups, but compute within them.

10.3.11.1 Computing With Group-wise Summaries

Let's make a new variable that is the years of life expectancy gained (lost) relative to 1952, for each individual country. We group by country and use `mutate()` to make a new variable. The `first()` function extracts the first value from a vector. Notice that `first()` is operating on the vector of life expectancies *within each country group*.

```
my_gap %>%
  group_by(country) %>%
  select(country, year, lifeExp) %>%
  mutate(lifeExp_gain = lifeExp - first(lifeExp)) %>%
  filter(year < 1963)
## # A tibble: 426 x 4
## # Groups:   country [142]
##   country year lifeExp lifeExp_gain
##   <fctr> <int> <dbl>     <dbl>
## 1 Afghanistan 1952 28.801     0.000
## 2 Afghanistan 1957 30.332     1.531
## 3 Afghanistan 1962 31.997     3.196
## 4  Albania 1952 55.230     0.000
## 5  Albania 1957 59.280     4.050
## 6  Albania 1962 64.820     9.590
## 7  Algeria 1952 43.077     0.000
## 8  Algeria 1957 45.685     2.608
## 9  Algeria 1962 48.303     5.226
## 10 Angola 1952 30.015     0.000
## # ... with 416 more rows
```

Within the country, we take the difference between life expectancy in year i and life expectancy in 1952. Therefore, we always see zeroes for 1952 and, for most countries, a sequence of positive and increasing numbers.

10.3.11.2 Window Functions

Window functions take n inputs and give back n outputs. Furthermore, the output depends on all the values. So `rank()` is a window function but `log()` is not. Here we use window functions based on ranks and offsets.

Let's revisit the worst and best life expectancies in Asia over time, but retaining info about *which* country contributes these extreme values.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year) %>%
  filter(min_rank(desc(lifeExp)) < 2 | min_rank(lifeExp) < 2) %>%
  arrange(year) %>%
  print(n = Inf)
## # A tibble: 24 x 3
## # Groups:   year [12]
##   year  country lifeExp
##   <int>   <fctr>   <dbl>
## 1 1952 Afghanistan 28.801
## 2 1952    Israel 65.390
## 3 1957 Afghanistan 30.332
## 4 1957    Israel 67.840
## 5 1962 Afghanistan 31.997
## 6 1962    Israel 69.390
## 7 1967 Afghanistan 34.020
## 8 1967     Japan 71.430
## 9 1972 Afghanistan 36.088
##10 1972     Japan 73.420
##11 1977  Cambodia 31.220
##12 1977     Japan 75.380
##13 1982 Afghanistan 39.854
##14 1982     Japan 77.110
##15 1987 Afghanistan 40.822
##16 1987     Japan 78.670
##17 1992 Afghanistan 41.674
##18 1992     Japan 79.360
##19 1997 Afghanistan 41.763
##20 1997     Japan 80.690
##21 2002 Afghanistan 42.129
##22 2002     Japan 82.000
##23 2007 Afghanistan 43.828
##24 2007     Japan 82.603
```

We see that (min = Afghanistan, max = Japan) is the most frequent result, but Cambodia and Israel pop up at least once each as the min or max, respectively. That table should make you

impatient for our upcoming work on tidying and reshaping data! Wouldn't it be nice to have one row per year?

How did that actually work? First, I store and view a partial that leaves off the `filter()` statement. All of these operations should be familiar.

```
asia <- my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  group_by(year)
asia
## # A tibble: 396 x 3
## # Groups:   year [12]
##   year  country lifeExp
##   <int>   <fctr>   <dbl>
## 1 1952 Afghanistan 28.801
## 2 1957 Afghanistan 30.332
## 3 1962 Afghanistan 31.997
## 4 1967 Afghanistan 34.020
## 5 1972 Afghanistan 36.088
## 6 1977 Afghanistan 38.438
## 7 1982 Afghanistan 39.854
## 8 1987 Afghanistan 40.822
## 9 1992 Afghanistan 41.674
## 10 1997 Afghanistan 41.763
## # ... with 386 more rows
```

Now we apply a window function — `min_rank()`. Since Asia is grouped by year, `min_rank()` operates within mini-datasets, each for a specific year. Applied to the variable `lifeExp`, `min_rank()` returns the rank of each country's observed life expectancy. FYI, the `min` part just specifies how ties are broken. Here is an explicit peek at these within-year life expectancy ranks, in both the (default) ascending and descending order.

For concreteness, I use `mutate()` to actually create these variables, even though I dropped this in the solution above. Let's look at a bit of that.

```
asia %>%
  mutate(le_rank = min_rank(lifeExp),
         le_desc_rank = min_rank(desc(lifeExp))) %>%
  filter(country %in% c("Afghanistan", "Japan", "Thailand"), year > 1995)
## # A tibble: 9 x 5
## # Groups:   year [3]
##   year  country lifeExp le_rank le_desc_rank
##   <int>   <fctr>   <dbl>   <int>     <int>
## 1 1997 Afghanistan 41.763     1         33
## 2 2002 Afghanistan 42.129     1         33
## 3 2007 Afghanistan 43.828     1         33
```

```
## 4 1997    Japan 80.690    33      1
## 5 2002    Japan 82.000    33      1
## 6 2007    Japan 82.603    33      1
## 7 1997    Thailand 67.521    12     22
## 8 2002    Thailand 68.564    12     22
## 9 2007    Thailand 70.616    12     22
```

Afghanistan tends to present 1s in the `le_rank` variable, Japan tends to present 1s in the `le_desc_rank` variable and other countries, like Thailand, present less extreme ranks. You can understand the original `filter()` statement now:

```
filter(min_rank(desc(lifeExp)) < 2 | min_rank(lifeExp) < 2)
```

These two sets of ranks are formed on the fly, within year group, and `filter()` retains rows with rank less than 2, which means ... the row with rank = 1. Since we do this for ascending and descending ranks, we get both the min and the max.

If we had wanted just the min OR the max, an alternative approach using `top_n()` would have worked.

```
my_gap %>%
  filter(continent == "Asia") %>%
  select(year, country, lifeExp) %>%
  arrange(year) %>%
  group_by(year) %>%
  #top_n(1, wt = lifeExp)    ## gets the min
  top_n(1, wt = desc(lifeExp)) ## gets the max
## # A tibble: 12 x 3
## # Groups:   year [12]
##   year  country lifeExp
##   <int>   <fctr>   <dbl>
## 1 1952 Afghanistan 28.801
## 2 1957 Afghanistan 30.332
## 3 1962 Afghanistan 31.997
## 4 1967 Afghanistan 34.020
## 5 1972 Afghanistan 36.088
## 6 1977  Cambodia 31.220
## 7 1982 Afghanistan 39.854
## 8 1987 Afghanistan 40.822
## 9 1992 Afghanistan 41.674
## 10 1997 Afghanistan 41.763
## 11 2002 Afghanistan 42.129
## 12 2007 Afghanistan 43.828
```

10.3.12 Grand Finale

So let's answer that "simple" question: Which country experienced the sharpest five-year drop in life expectancy? Recall that this excerpt of the Gapminder data only have data every five years (e.g., for 1952, 1957, etc.). So, this really means looking at life expectancy changes between adjacent time points.

At this point that's just too easy, so let's do it by continent while we're at it.

```
my_gap %>%
  select(country, year, continent, lifeExp) %>%
  group_by(continent, country) %>%
  ## within country, take (lifeExp in year i) - (lifeExp in year i - 1)
  ## positive means lifeExp went up, negative means it went down
  mutate(le_delta = lifeExp - lag(lifeExp)) %>%
  ## within country, retain the worst lifeExp change = smallest or most negative
  summarize(worst_le_delta = min(le_delta, na.rm = TRUE)) %>%
  ## within continent, retain the row with the lowest worst_le_delta
  top_n(-1, wt = worst_le_delta) %>%
  arrange(worst_le_delta)
## # A tibble: 5 x 3
## # Groups:   continent [5]
##   continent    country worst_le_delta
##   <fctr>      <fctr>      <dbl>
## 1 Africa      Rwanda      -20.421
## 2 Asia        Cambodia    -9.097
## 3 Americas    El Salvador  -1.511
## 4 Europe      Montenegro  -1.464
## 5 Oceania     Australia    0.170
```

Ponder that for a while. The subject matter and the code. Mostly you're seeing what genocide looks like in dry statistics on average life expectancy.

Break the code into pieces, starting at the top, and inspect the intermediate results. That's certainly how I was able to *write* such a thing. These commands do not [leap fully formed out of anyone's forehead](#) — they are built up gradually, with lots of errors and refinements along the way. I'm not even sure it's a great idea to do so much manipulation in one fell swoop. Is the statement above really hard for you to read? If yes, then, by all means, break it into pieces and make some intermediate objects. Your code should be easy to write and read when you're done.

10.3.13 Working With Two Datasets

When working with two or more datasets, we routinely join them. dplyr adopts the SQL-dialect for joining datasets. Below are examples for those of us who don't speak SQL so well. There are lots of [Venn diagrams re SQL joins on the interwebs](#), but I wanted R examples.

[Full documentation](#) for the dplyr package, which is developed by Hadley Wickham and Romain Francois on [GitHub](#). The [vignette on Two-table verbs](#) covers the joins shown here.

10.3.14 Variable Recoding With dplyr

- recode and recode_factor: Replace numeric values based on their position, and character values by their name;
- if_else: Replace values based on a logical vector; and
- case_when: Vectorise multiple if and else if statements.

Recoding, when to use which function: — one-to-one, many-to-one: recode and recode_factor
Download the NHTS 2009 data file for the demos [here](#) (Right click and select Save As...)

```
library(tidyverse)

# load NHTS2009 travel diaries subset
dd <- read_csv("data/NHTS2009_dd.csv")

# recode race (HH_RACE column) according to data dictionary: http://nhts.ornl.gov/tables09/C
odebookPage.aspx?id=951
dd %>% mutate(hh_race_str=recode(HH_RACE,
                                "01"="White",
                                "02"="African American, Black",
                                "03"="Asian Only",
                                "04"="American Indian, Alaskan Native",
                                "05"="Native Hawaiian, other Pacific",
                                "06"="Multiracial",
                                "07"="Hispanic/Mexican",
                                "97"="Other specify",
                                .default = as.character(NA) # any unspecified values would be assigned NA
                                )) %>%

select(HH_RACE, hh_race_str)
## # A tibble: 304 x 2
##   HH_RACE hh_race_str
##   <chr>    <chr>
## 1     01    White
## 2     01    White
## 3     01    White
## 4     01    White
## 5     01    White
## 6     01    White
## 7     01    White
## 8     01    White
## 9     01    White
## 10    01    White
## # ... with 294 more rows
```

- a logical condition: if_else

```
# code driving & non-driving based on travel modes (TRPTRANS column) data dictionary: http://nhts.ornl.gov/tables09/CodebookPage.aspx?id=1084
dd %>% mutate(driving=ifelse(TRPTRANS %in% c("01", "02", "03", "04", "05", "06", "07"),
1, 0),
  driving=ifelse(TRPTRANS %in% c("-1", "-7", "-8", "-9"), NA, driving) # retain missin
g values as NA
) %>%
select(TRPTRANS, driving)
## # A tibble: 304 x 2
##   TRPTRANS driving
##   <chr>   <dbl>
## 1     03     1
## 2     03     1
## 3     03     1
## 4     03     1
## 5     03     1
## 6     03     1
## 7     03     1
## 8     03     1
## 9     03     1
## 10    03     1
## # ... with 294 more rows
```

- multiple logical conditions: case_when

```
# code driving & non-driving based on travel modes (TRPTRANS column) data dictionary: http://nhts.ornl.gov/tables09/CodebookPage.aspx?id=1084 use case_when
dd %>% mutate(driving=case_when(
  TRPTRANS %in% c("01", "02", "03", "04", "05", "06", "07") ~ 1,
  TRPTRANS %in% c("-1", "-7", "-8", "-9") ~ as.double(NA), # retain missing values as NA
  TRUE ~ 0)) %>%
select(TRPTRANS, driving)
## # A tibble: 304 x 2
##   TRPTRANS driving
##   <chr>   <dbl>
## 1     03     1
## 2     03     1
## 3     03     1
## 4     03     1
## 5     03     1
## 6     03     1
## 7     03     1
## 8     03     1
## 9     03     1
## 10    03     1
## # ... with 294 more rows
```

reclassify households into low, med, high income based on HHFAMINC column data dictionary: <http://nhts.ornl.gov/tables09/CodebookPage.aspx?id=949> with brackets [0, 30000, 6000]

```
dd <- dd %>% mutate(income_cat=case_when(
  HHFAMINC %in% c("01", "02", "03", "04", "05", "06") ~ "low income",
  HHFAMINC %in% c("07", "08", "09", "10", "11", "12") ~ "med income",
  HHFAMINC %in% c("13", "14", "15", "16", "17", "18") ~ "high income",
  TRUE ~ as.character(NA) # retain missing values as NA
))
```

verify recoding results with group_by & tally

```
dd %>% group_by(HHFAMINC, income_cat) %>%
  tally()
```

```
## # A tibble: 13 x 3
```

```
## # Groups:   HHFAMINC [?]
```

```
##   HHFAMINC income_cat    n
```

```
##   <chr>      <chr> <int>
```

```
## 1    01 low income     4
```

```
## 2    02 low income     2
```

```
## 3    03 low income    12
```

```
## 4    04 low income     2
```

```
## 5    06 low income    18
```

```
## 6    07 med income     6
```

```
## 7    08 med income    10
```

```
## 8    12 med income     7
```

```
## 9    14 high income    20
```

```
## 10   16 high income    38
```

```
## 11   17 high income    64
```

```
## 12   18 high income   115
```

```
## 13   -7      <NA>      6
```

10.3.15 Programming With dplyr

- [Non-standard Evaluation in dplyr 0.6+](#)
- [Programming with dplyr](#)

10.4 EXERCISE

1. Filter days where there are missing values in bike counts and weather information. Count the number of days with missing values on either bike counts or weather information.
2. Calculate weekly, monthly, and annual bike counts from the daily bike counts data using dplyr verbs.
3. Join the bike counts data with the weather data. Which type of joins work best here?
4. With the [NHTS2009 travel diaries data](#), how do you calculate total miles traveled (using any modes) and miles traveled by driving for each household (hint: the TRPMILES column contains information about trip distance for each member of a household).

5. [Challenge] How do you compute the average household-level miles driving per capita by income categories (low, med, high)?

10.5 LEARNING MORE

1. [Dataframe Manipulation with dplyr](#)
2. [Data Wrangling Cheatsheet](#)
3. [Hands-on dplyr tutorial for faster data manipulation in R](#) by Data School, which includes a link to an R Markdown document and links to videos.

11.0 DATA VISUALIZATION WITH GGPLOT2

11.1 READINGS

1. [R4DS Chapter 3 Data Visualization](#)
2. [R4DS Chapter 7 Exploratory Data Analysis](#)

11.2 OVERVIEW

ggplot2 is a grammar of graphics, a coherent system for describing and building graphs.

- data (data frame): `ggplot(dataset)`
- aesthetic mapping (coordination system, color, size, line type, position etc): `aes()`
- geometries: `geom_point`, `geom_line`, `geom_bar`, ...
- stat (summarization/transformation of data): `stat_smooth`,
- scale (map data values into computer values): `scale_y_continuous()`, `scale_x_log10`, ...
- facet: `facet_wrap`, `facet_grid` and
- fine tuning coordinates, theme, labels: `coordinate_*`, `theme_bw`, `labs(x=..., y=...)`

11.3 TIPS

- Keep your data tidy (three rules);
- Reshape your data (tall and thin often better for visualization and analysis); and
- [Use factors \(for reorder and better labeling\)](#).

11.4 LESSON

Lesson below is adapted from [R4DS Chapter 3 Data Visualization](#).

11.4.1 Prerequisites

This session focuses on ggplot2, one of the core members of the tidyverse. To access the datasets, help pages, and functions that we will use in this chapter, load the tidyverse by running this code:

```
library(tidyverse)
library(ggplot2)
```

11.4.2 First Steps

Let's use our first graph to answer a question: Do cars with big engines use more fuel than cars with small engines? You probably already have an answer, but try to make your answer precise. What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

11.4.3 The mpg Data Frame

You can test your answer with the mpg **data frame** found in ggplot2 (aka ggplot2::mpg). A data frame is a rectangular collection of variables (in the columns) and observations (in the rows). mpg contains observations collected by the U.S. Environment Protection Agency on 38 models of cars.

```
mpg
## # A tibble: 234 x 11
##   manufacturer    model displ  year  cyl  trans  drv  cty  hwy
##   <chr>         <chr> <dbl> <int> <int> <chr> <chr> <int> <int>
## 1      audi      a4    1.8  1999   4 auto(l5) f   18  29
## 2      audi      a4    1.8  1999   4 manual(m5) f   21  29
## 3      audi      a4    2.0  2008   4 manual(m6) f   20  31
## 4      audi      a4    2.0  2008   4 auto(av) f   21  30
## 5      audi      a4    2.8  1999   6 auto(l5) f   16  26
## 6      audi      a4    2.8  1999   6 manual(m5) f   18  26
## 7      audi      a4    3.1  2008   6 auto(av) f   18  27
## 8      audi a4 quattro  1.8  1999   4 manual(m5) 4   18  26
## 9      audi a4 quattro  1.8  1999   4 auto(l5) 4   16  25
## 10     audi a4 quattro  2.0  2008   4 manual(m6) 4   20  28
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

Among the variables in mpg are:

1. displ, a car's engine size, in liters.
2. hwy, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with low fuel efficiency consumes more fuel than a car with high fuel efficiency when they travel the same distance.

To learn more about mpg, open its help page by running ?mpg.

11.4.4 Creating a ggplot

To plot mpg, run this code to put displ on the x-axis and hwy on the y-axis:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```

Figure 11.1 Scatter plot of highway mpg by displ

The plot shows a negative relationship between engine size (displ) and fuel efficiency (hwy). In other words, cars with big engines use more fuel. Does this confirm or refute your hypothesis about fuel efficiency and engine size?

With `ggplot2`, you begin a plot with the function `ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument of `ggplot()` is the dataset to use in the graph. So `ggplot(data = mpg)` creates an empty graph, but it's not very interesting so I'm not going to show it here.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatter plot. `ggplot2` comes with many `geom` functions that each add a different type of layer to a plot. You'll learn a whole bunch of them throughout this chapter.

Each `geom` function in `ggplot2` takes a mapping argument. This defines how variables in your dataset are mapped to visual properties. The mapping argument is always paired with `aes()`, and the `x` and `y` arguments of `aes()` specify which variables to map to the `x`- and `y`-axes. `ggplot2` looks for the mapped variable in the `data` argument, in this case, `mpg`.

11.4.5 A Graphing Template

Let's turn this code into a reusable template for making graphs with ggplot2. To make a graph, replace the bracketed sections in the code below with a dataset, a geom function, or a collection of mappings.

```
ggplot(data = <DATA>) +  
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The rest of this session will show you how to complete and extend this template to make different types of graphs. We will begin with the <MAPPINGS> component.

11.4.6 Aesthetic Mappings

"The greatest value of a picture is when it forces us to notice what we never expected to see." — John Tukey

In the plot below, one group of points (highlighted in red) seems to fall outside of the linear trend. These cars have a higher mileage than you might expect. How can you explain these cars?

Figure 11.2 Scatter plot with some cars highlighted in red

Let's hypothesize that the cars are hybrids. One way to test this hypothesis is to look at the class value for each car. The class variable of the mpg dataset classifies cars into groups such as

compact, midsize, and SUV. If the outlying points are hybrids, they should be classified as compact cars or, perhaps, subcompact cars (keep in mind that these data were collected before hybrid trucks and SUVs became popular).

You can add a third variable, like class, to a two-dimensional scatter plot by mapping it to an **aesthetic**. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points.

You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset. For example, you can map the colors of your points to the class variable to reveal the class of each car.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

Figure 11.3 Scatter plot of highway mpg by displ, with class of cars differentiated using color

To map an aesthetic to a variable, associate the name of the aesthetic to the name of the variable inside `aes()`. `ggplot2` will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as **scaling**. `ggplot2` will also add a legend that explains which levels correspond to which values.

The colors reveal that many of the unusual points are two-seater cars. These cars don't seem like hybrids, and are, in fact, sports cars! Sports cars have large engines like SUVs and pickup trucks,

but small bodies like midsize and compact cars, which improves their gas mileage. In hindsight, these cars were unlikely to be hybrids since they have large engines.

In the above example, we mapped class to the color aesthetic, but we could have mapped class to the size aesthetic in the same way. In this case, the exact size of each point would reveal its class affiliation. We get a *warning* here because mapping an unordered variable (class) to an ordered aesthetic (size) is not a good idea.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
```

Figure 11.4 Scatter plot of highway mpg by displ, with class of cars differentiated using size

Or we could have mapped class to the *alpha* aesthetic, which controls the transparency of the points, or the shape of the points.

```
# Top  
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, alpha = class))  
  
# Bottom  
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

Figure 11.4 Scatter plot of highway mpg by displ, with class of cars differentiated using size

What happened to the SUVs? ggplot2 will only use six shapes at a time. By default, additional groups will go unplotted when you use the shape aesthetic.

For each aesthetic, you use `aes()` to associate the name of the aesthetic with a variable to display. The `aes()` function gathers together each of the aesthetic mappings used by a layer and passes them to the layer's mapping argument. The syntax highlights a useful insight about x and y: The x and y locations of a point are themselves aesthetics, visual properties that you can map to variables to display information about the data.

Once you map an aesthetic, ggplot2 takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a legend that explains the mapping between levels and values. For x and y aesthetics, ggplot2 does not create a legend, but it creates an axis line with tick marks and a label. The axis line acts as a legend; it explains the mapping between locations and values.

You can also *set* the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```

Figure 11.5 Scatter plot with color aesthetic manually set to blue

11.4.7 Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into **facets**, subplots that each display one subset of the data.

To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with `~` followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation"). The variable that you pass to `facet_wrap()` should be discrete.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_wrap(~ class, nrow = 2)
```

Figure 11.6 Scatter plots of highway mpg by displ, faceted by class

To facet your plot on the combination of two variables, add `facet_grid()` to your plot call. The first argument of `facet_grid()` is also a formula. This time, the formula should contain two variable names separated by a `~`.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_grid(drv ~ cyl)
```

Figure 11.6 Scatter plots of highway mpg by displ, faceted by class

If you prefer to not facet in the rows or columns dimension, use a . instead of a variable name (e.g., `+ facet_grid(. ~ cyl)`).

11.4.8 Geometric Objects

How are these two plots similar?

Figure 11.7 Scatter plot and smooth curve of highway mpg by displ

Both plots contain the same x and y variables, and both describe the same data. But the plots are not identical. Each uses a different visual object to represent the data. In ggplot2 syntax, we say that they use different **geoms**.

A **geom** is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatter plots break the trend; they use the point geom. As we see above, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to ggplot(). For instance, to make the plots above, you can use this code:

```
# top
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

# bottom
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Every geom function in ggplot2 takes a mapping argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn't set the "shape" of a line. On the other hand, you *could* set the linetype of a line. geom_smooth() will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype.

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```

Figure 11.8 Smooth curves of highway mpg by displ, grouped by drv

Here `geom_smooth()` separates the cars into three lines based on their `drv` value, which describes a car's drivetrain. One line describes all of the points with a 4 value, one line describes all of the points with an f value, and one line describes all of the points with an r value. Here, 4 stands for four-wheel drive, f for front-wheel drive, and r for rear-wheel drive.

If this sounds strange, we can make it more clear by overlaying the lines on top of the raw data and then coloring everything according to `drv`.

Figure 11.9 Scatter plot and smooth curves of highway mpg by displ, grouped by drv

Notice that this plot contains two geoms in the same graph! If this makes you excited, buckle up. In the next section, we will learn how to place multiple geoms in the same plot.

ggplot2 provides over 30 geoms, and extension packages provide even more (see <https://www.ggplot2-exts.org> for a sampling). The best way to get a comprehensive overview is the ggplot2 cheatsheet, which you can find at <https://www.rstudio.com/resources/cheatsheets/>. To learn more about any single geom, use help: `?geom_smooth`.

Many geoms, like `geom_smooth()`, use a single geometric object to display multiple rows of data. For these geoms, you can set the group aesthetic to a categorical variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable. In practice, ggplot2 will automatically group the data for these geoms whenever you map an aesthetic to a discrete variable (as in the linetype example). It is convenient to rely on this feature because the group aesthetic by itself does not add a legend or distinguishing features to the geoms.

```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy))  
  
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))
```

Figure 11.10 Smooth curves of highway mpg by displ

To display multiple geoms in the same plot, add multiple geom functions to `ggplot()`:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Figure 11.11 Scatter plot of highway mpg by displ super-imposed with `stat_smooth`

This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display `cty` instead of `hwy`. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to `ggplot()`. `ggplot2` will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +  
  geom_point() +  
  geom_smooth()
```

11.4.9 Statistical Transformations

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the total number of diamonds in the `diamonds` dataset, grouped by `cut`. The `diamonds` dataset comes in `ggplot2` and contains information about ~54,000 diamonds,

including the price, carat, color, clarity, and cut of each diamond. The chart shows that more diamonds are available with high-quality cuts than with low-quality cuts.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut))
```

Figure 11.12 Bar chart of diamonds by cut

On the x-axis, the chart displays cut, a variable for diamonds. On the y-axis, it displays count, but count is not a variable for diamonds! Where does count come from? Many graphs, like scatter plots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- Bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
- Smoothers fit a model to your data and then plot predictions from the model.
- Boxplots compute a robust summary of the distribution and then display a specially formatted box.

The algorithm used to calculate new values for a graph is called a **stat**, short for statistical transformation. The figure below describes how this process works with `geom_bar()`.

1. `geom_bar()` begins with the **diamonds** data set

2. `geom_bar()` transforms the data with the "count" stat, which returns a data set of cut values and counts.

3. `geom_bar()` uses the transformed data to build the plot. cut is mapped to the x axis, count is mapped to the y axis.

Figure 11.13 Process behind `geom_bar`

You can learn which stat a geom uses by inspecting the default value for the stat argument. For example, `?geom_bar` shows the default value for stat is "count," which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`, and if you scroll down you can find a section called "computed variables." That tells that it computes two new variables: count and prop.

You can generally use geoms and stats interchangeably. For example, you can recreate the previous plot using `stat_count()` instead of `geom_bar()`:

```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```

Figure 11.14 Bar chart of diamonds by cut created with `stat_count`

This works because every geom has a default stat, and every stat has a default geom. This means that you can typically use geoms without worrying about the underlying statistical transformation. There are three reasons you might need to use a stat explicitly:

1. You might want to override the default stat. In the code below, I change the stat of `geom_bar()` from `count` (the default) to `identity`. This lets me map the height of the bars to the raw values of a `y` variable. Unfortunately, when people talk about bar charts casually, they might be referring to this type of bar chart, where the height of the bar is already present in the data, or the previous bar chart, where the height of the bar is generated by counting rows.

```
demo <- tribble(
  ~a, ~b,
  "bar_1", 20,
  "bar_2", 30,
  "bar_3", 40
)

ggplot(data = demo) +
  geom_bar(mapping = aes(x = a, y = b), stat = "identity")
```

Figure 11.15 Bar chart created with `stat=identity`

2. You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count:

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
```

Figure 11.16 Bar chart with y-axis in proportion

To find the variables computed by the stat, look for the help section titled "computed variables."

3. You might want to draw greater attention to the statistical transformation in your code. For example, you might use `stat_summary()`, which summarises the y values for each unique x value, to draw attention to the summary that you're computing:

```
ggplot(data = diamonds) +  
  stat_summary(  
    mapping = aes(x = cut, y = depth),  
    fun.ymin = min,  
    fun.ymax = max,  
    fun.y = median  
  )
```

Figure 11.17 Chart with stat-summary

ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in a usual way (e.g., `?stat_bin`). To see a complete list of stats, try the ggplot2 cheatsheet.

11.4.10 Position Adjustments

There's one more piece of magic associated with bar charts. You can color a bar chart using either the color aesthetic, or more usefully, fill:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, colour = cut))
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```

Figure 11.18 Bar chart with fill and color

Note what happens if you map the fill aesthetic to another variable, like clarity: the bars are automatically stacked. Each colored rectangle represents a combination of cut and clarity.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity))
```

Figure 11.19 Stacked bar chart of diamonds by cut and clarity

The stacking is performed automatically by the **position adjustment** specified by the position argument. If you don't want a stacked bar chart, you can use one of three other options: "identity," "dodge" or "fill,"

- position = "identity" will place each object exactly where it falls in the context of the graph. This is not very useful for bars because it overlaps them. To see that overlapping we either need to make the bars slightly transparent by setting alpha to a small value or completely transparent by setting fill = NA.

```
ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) +  
  geom_bar(alpha = 1/5, position = "identity")  
ggplot(data = diamonds, mapping = aes(x = cut, colour = clarity)) +  
  geom_bar(fill = NA, position = "identity")
```

Figure 11.20 Bar chart with position='identity'

The identity position adjustment is more useful for 2d geoms, like points, where it is the default.

- `position = "fill"` works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
```

Figure 11.21 Bar chart with position='fill'

- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
```

Figure 11.22 Bar chart with position='identity'

There's one other type of adjustment that's not useful for bar charts, but it can be very useful for scatter plots. Recall our first scatter plot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?

Figure 11.23 Scatter plot of highway mpg by displ

The values of `hwy` and `displ` are rounded so the points appear on a grid and many points overlap each other. This problem is known as **overplotting**. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination of `hwy` and `displ` that contains 109 values?

You can avoid this gridding by setting the position adjustment to "jitter." `Position = "jitter"` adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy), position = "jitter")
```

Figure 11.24 Scatter plot of highway mpg by displ, with position='jitter'

Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, ggplot2 comes with a shorthand for `geom_point(position = "jitter")`: `geom_jitter()`.

To learn more about a position adjustment, look up the help page associated with each adjustment: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter`, and `?position_stack`.

11.4.11 Coordinate Systems

Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y position act independently to find the location of each point. There are a number of other coordinate systems that are occasionally helpful.

- `coord_flip()` switches the x- and y-axis. This is useful if, for example, you want horizontal boxplots. It's also useful for long labels: It's hard to get them to fit without overlapping on the x-axis.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +  
geom_boxplot()
```

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

Figure 11.24 Bar chart with flipped - and y-axis

- `coord_quickmap()` sets the aspect ratio correctly for maps. This is very important if you're plotting spatial data with `ggplot2`.

```
nz <- map_data("nz")
```

```
ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")
```

```
ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black") +
  coord_quickmap()
```

Figure 11.25 Map of New Zealand with ggplot2

- `coord_polar()` uses polar coordinates. Polar coordinates reveal an interesting connection between a bar chart and a Coxcomb chart.

```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
bar + coord_polar()
```

Figure 11.26 Coxcomb chart with the flipped and polar coordinate system

11.4.12 The Layered Grammar of Graphics

In the previous sections, you learned much more than how to make scatter plots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with ggplot2. To see this, let's add position adjustments, stats, coordinate systems, and faceting to our code template:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(  
    mapping = aes(<MAPPINGS>),  
    stat = <STAT>,  
    position = <POSITION>  
  ) +  
  <COORDINATE_FUNCTION> +  
  <FACET_FUNCTION>
```

Our new template takes seven parameters, the bracketed words that appear in the template. In practice, you rarely need to supply all seven parameters to make a graph because ggplot2 will provide useful defaults for everything except the data, the mappings, and the geom function. The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system, and a faceting scheme.

To see how this works, consider how you could build a basic plot from scratch: You could start with a dataset and then transform it into the information that you want to display (with a stat).

1. Begin with the **diamonds** data set

2. Compute counts for each cut value with **stat_count()**.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

Figure 11.27 Visualization Grammar Step 1

Next, you could choose a geometric object to represent each observation in the transformed data. You could then use the aesthetic properties of the geoms to represent variables in the data. You would map the values of each variable to the levels of an aesthetic.

3. Represent each observation with a bar.

4. Map the **fill** of each bar to the **..count..** variable.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

Figure 11.28 Visualization Grammar Step 2

You'd then select a coordinate system to place the geoms into. You'd use the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables. At that point, you would have a complete graph, but you could further adjust the positions of the geoms within the coordinate system (a position adjustment) or split the graph into subplots (faceting). You could also extend the plot by adding one or more additional layers, where each additional layer uses a dataset, a geom, a set of mappings, a stat, and a position adjustment.

Figure 11.29 Visualization Grammar Step 3

You could use this method to build *any* plot that you imagine. In other words, you can use the code template that you've learned in this chapter to build hundreds of thousands of unique plots.

11.4.13 Additional Resources

1. [ggfortify - plot some popular R packages with ggplot2](#)
2. [geom_map](#)
3. [geom_sf - ggplot2 working with simple feature](#) and the [simple feature package](#) for geospatial analysis
4. [ggmap - raster map tiles from popular online mapping services](#)
5. [interactive maps and charts with R](#)

11.5 EXERCISE

1. Plot the daily bike counts for Hawthorne and Tilikum with ggplot2; experiment with different options and select the one works best for you;
2. Plot weekly, monthly, and annual bike counts for Hawthorne and Tilikum;
3. Now think about and try out different ways to plot weather information along with your daily bike counts (separately or together); and
4. You can combine time series stats and ggplot2/ggfortify to plot seasonal variation, trend and noise separately. See [here](#) for an example of how to do this.

11.6 LEARNING MORE:

1. [ggplot2 tutorial by Prof. Jenny Bryan](#)
2. [R4DS Chapter 28. Graphics for communication](#)
3. [The ggplot2 Cheatsheet - RStudio](#)
4. [R Graphics Cookbook](#)

12.0 SPLIT-APPLY-COMBINE PATTERN IN DATA MODELING

12.1 READINGS

1. [Wickham, Hadley, 2011. The Split-Apply-Combine Strategy for Data Analysis, Journal of Statistical Software, Volume 40, Issue 1.](#)
2. [R4DS Chapter 25. Many Models](#)

If you are not familiar with the basic workflow of fitting models with R, you should review two chapters:

1. [R4DS Chapter 23. Model basics](#)
2. [R4DS Chapter 24. Model building](#)

12.2 SPLIT-APPLY-COMBINE

A common analytical pattern is to:

- split data into pieces,
- apply some function to each piece, and
- combine the results back together again.

Generally, avoid using loops when you need to do Split-Apply-Combine, consider these alternatives instead:

1. Entry level: `dplyr::group_by()`
2. General approach: nesting
3. *apply functions and plyr package (non-tidyverse solution)

12.3 LESSON

The lesson below is adapted from [UBC's STAT 545 by Professor Jenny Bryan](#).

12.3.1 Think Before You Create Excerpts of Your Data ...

If you feel the urge to store a little snippet of your data:

```
snippet <- my_big_dataset %>%  
  filter(some_variable == some_value)  
  
## or  
snippet <- subset(my_big_dataset, some_variable == some_value)
```

Stop and ask yourself ...

Do I want to create mini datasets for each level of some factor (or a unique combination of several factors) ... in order to compute or graph something? If YES, **use proper data aggregation techniques** or facetting in `ggplot2` -- **don't subset the data**. Or, more realistically,

subset the data only as a temporary measure while you develop your elegant code for computing on or visualizing these data subsets.

If NO, then maybe you really do need to store a copy of a subset of the data. But seriously consider whether one of these strategies that subset at "compute time" will get the job done:

- Pass an intact data frame and use the `subset =` argument of a command. Many functions have it!

```
## regress life expectancy on year for Canada
canada_fit <-
  lm(lifeExp ~ year, data = gapminder, subset = country == "Canada")
```

- Pipe filtered data into a command. A very general solution.

```
## compare gdpPercap in Australia vs New Zealand
oceania_ttest <- gapminder %>%
  filter(continent == "Asia") %>%
  t.test(gdpPercap ~ country, data = .)
```

Creating copies and excerpts of your data clutters up your workspace, script and mind, often leading to confusion and mistakes. It can be useful during development, but try to eliminate this as you polish your analysis.

12.3.2 Prerequisite

Load tidyverse and gapminder:

```
library(gapminder)
library(tidyverse)
```

12.3.3 Split-Apply-Combine

A common analytical pattern is to

- split data into pieces,
- apply some function to each piece, and
- combine the results back together again.

R has specialized tools for this that are much nicer than whatever DIY approach you might be contemplating. Especially if it involves `for()` loops. There's nothing inherently wrong or evil about `for()` loops, but you will spend a lot of time and characters on bookkeeping tasks. That can be a great way to pilot a computation, because it is so blessedly concrete. But consider revisiting your implementation with higher-level data aggregation tools once you've established proof-of-principle.

In base R, these are the "apply" functions, such as `apply()`, `aggregate()`, `tapply()`, and `by()`. We prefer to use similar tools from the tidyverse, due to a more consistent interface, analysis-friendly output, and a more concise way to describe what to compute. The Hadley Wickham article in the

readings — the split-apply-combine strategy for data analysis — provides a good high-level overview, but know that the tidyverse approaches outlined here now supersede the plyr package described there.

12.3.4 Entry Level: `dplyr::group_by()`

The most lightweight solution is given by `dplyr::group_by()`. `group_by()` adds some extra grouping structure to a data frame, based on levels of one or more categorical variables, but leaves the data frame intact. Then you can do group-wise computations using various functions in the tidyverse that automatically honor these groups.

The main function here is `summarize()`, which collapses each group into a single row in a new group-summarized data frame.

When does this break down? If anything you want to compute for a group is more complicated than, say, a single number, you have outgrown `dplyr::summarize()`. Example: Getting the range of life expectancies seen for each continent doesn't work with the code below.

```
gapminder %>%  
  group_by(continent) %>%  
  summarize(range = range(lifeExp))
```

Now, in this case, we could reframe this in a `summarize()`-friendly form, but that doesn't work in general.

```
gapminder %>%  
  group_by(continent) %>%  
  summarize_each(funs(min, max), lifeExp)  
## # A tibble: 5 x 3  
##   continent lifeExp_min lifeExp_max  
##   <fctr>      <dbl>      <dbl>  
## 1 Africa      23.599      76.442  
## 2 Americas    37.579      80.653  
## 3 Asia        28.801      82.603  
## 4 Europe      43.585      81.757  
## 5 Oceania     69.120      81.235
```

12.3.5 General Approach: Nesting

We use **nesting** as an extension of **grouping** in order to support more general group-wise computation. The collapse to a single row per group happens right away here, unlike the simple grouping above. When you nest data, the non-grouping variables are packaged into group-specific data frames that are held in a special variable called a list-column. You then apply your computation to the components of this list (i.e., the data frames). List-columns take a little getting used to, but the payoff is huge.

Let's get ready to fit a model for each country in the Gapminder dataset. First, we group, as above, and then nest. We group by country and continent in order to keep those two variables "on the outside."

```
(gap_nested <- gapminder %>%
  group_by(continent, country) %>%
  nest())
## # A tibble: 142 x 3
##   continent    country      data
##   <fctr>      <fctr>    <list>
## 1   Asia Afghanistan <tibble [12 x 4]>
## 2  Europe   Albania <tibble [12 x 4]>
## 3  Africa   Algeria <tibble [12 x 4]>
## 4  Africa   Angola <tibble [12 x 4]>
## 5 Americas Argentina <tibble [12 x 4]>
## 6 Oceania  Australia <tibble [12 x 4]>
## 7  Europe   Austria <tibble [12 x 4]>
## 8   Asia    Bahrain <tibble [12 x 4]>
## 9   Asia    Bangladesh <tibble [12 x 4]>
## 10 Europe   Belgium <tibble [12 x 4]>
## # ... with 132 more rows
```

What do you notice?

- The immediate collapse to 142 rows, one per country.
- The familiar presence of continent and country.
- The unfamiliar look of the new data variable, which is ... a list! It is a list-column of continent-specific tibbles.

How on earth do you inspect it?!? In RStudio, `gap_nested %>% View()` is often helpful (moderately so in this case). It's often nicer to inspect a single element like so:

```
gap_nested[[1, "data"]]
## # A tibble: 12 x 4
##   year lifeExp  pop gdpPercap
##   <int> <dbl> <int> <dbl>
## 1 1952 28.801 8425333 779.4453
## 2 1957 30.332 9240934 820.8530
## 3 1962 31.997 10267083 853.1007
## 4 1967 34.020 11537966 836.1971
## 5 1972 36.088 13079460 739.9811
## 6 1977 38.438 14880372 786.1134
## 7 1982 39.854 12881816 978.0114
## 8 1987 40.822 13867957 852.3959
## 9 1992 41.674 16317921 649.3414
## 10 1997 41.763 22227415 635.3414
## 11 2002 42.129 25268405 726.7341
## 12 2007 43.828 31889923 974.5803
```

Here double square brackets are used to extract a single element and they are simplifying. An equivalent call is `gap_nested[["data"]][[1]]` (which I find even more opaque) or `gap_nested$data[[1]]` (which is pretty nice). We're looking at the first of 142 country-specific data frames, which happen to be for Afghanistan.

The presence of list-columns is always a temporary, uncomfortable state.

12.3.6 Apply a Function `purrr::map()` and `mutate()`

How do we now iterate over the elements of `gap_nested$data`? It is a list, so we use general approaches for applying a function to each element of a list. In base R, this means `lapply()`, but we will use the tidyverse function `purrr::map()`, which has a few advantages [described elsewhere](#). Walk before you run (i.e., do an example first)!

Let's fit a model to the data from Afghanistan. The form of the right-hand side is so that our intercept has a nice interpretation.

```
(fit <- lm(lifeExp ~ I(year - 1950), data = gap_nested[[1, "data"]]))
##
## Call:
## lm(formula = lifeExp ~ I(year - 1950), data = gap_nested[[1,
##   "data"]])
##
## Coefficients:
##   (Intercept) I(year - 1950)
##      29.3566      0.2753
```

Let's capture that logic in a function and informally test that it returns the same results for Afghanistan.

```
le_vs_yr <- function(df) {
  lm(lifeExp ~ I(year - 1950), data = df)
}

le_vs_yr(gap_nested[[1, "data"]])
##
## Call:
## lm(formula = lifeExp ~ I(year - 1950), data = df)
##
## Coefficients:
##   (Intercept) I(year - 1950)
##      29.3566      0.2753
```

After you walk, jog before you run. Use `purrr::map()` to apply the fitting function `le_vs_yr()` to the first two elements of `gap_nested$data`.

```
fits <- purrr::map(gap_nested$data[1:2], le_vs_yr)

fits
## [[1]]
##
## Call:
## lm(formula = lifeExp ~ I(year - 1950), data = df)
##
## Coefficients:
## (Intercept) I(year - 1950)
## 29.3566      0.2753
##
##
## [[2]]
##
## Call:
## lm(formula = lifeExp ~ I(year - 1950), data = df)
##
## Coefficients:
## (Intercept) I(year - 1950)
## 58.5598      0.3347
```

So, how do we run (i.e., scale this up to all countries)? And where do we put these fitted models? We'll use `purrr::map()` inside `mutate()`, meaning we store the models inside `gap_nested` in another list-column.

```
(gap_nested <- gap_nested %>%
  mutate(fit = purrr::map(data, le_vs_yr)))
## # A tibble: 142 x 4
##   continent country      data fit
##   <fctr>    <fctr>    <list> <list>
## 1 Asia Afghanistan <tibble [12 x 4]> <S3: lm>
## 2 Europe Albania <tibble [12 x 4]> <S3: lm>
## 3 Africa Algeria <tibble [12 x 4]> <S3: lm>
## 4 Africa Angola <tibble [12 x 4]> <S3: lm>
## 5 Americas Argentina <tibble [12 x 4]> <S3: lm>
## 6 Oceania Australia <tibble [12 x 4]> <S3: lm>
## 7 Europe Austria <tibble [12 x 4]> <S3: lm>
## 8 Asia Bahrain <tibble [12 x 4]> <S3: lm>
## 9 Asia Bangladesh <tibble [12 x 4]> <S3: lm>
## 10 Europe Belgium <tibble [12 x 4]> <S3: lm>
## # ... with 132 more rows
```

We have a new list-column, `fit`, that is even more intimidating than the previous, `data`. The `fit` variable holds 142 fitted linear models. What shall we do with that?

12.3.7 Always Have an Exit Strategy

The list-column state is an uncomfortable and temporary one. The goal is always to pull information out of these complicated objects and package it as something simpler, usually a tibble. You always want a list-column exit strategy!

We are not afraid to fit linear models in this example, because of a fantastic package from David Robinson, [broom](#), that provides exactly this for `lm()` and many other functions.

`broom` provides three key functions that take a `lm()` fit as input and give a useful tibble back:

- `tidy()`: a tidy version of `summary()` (e.g., a table with one row per parameter estimate)
- `augment()`: the original data, augmented with columns such as fitted values and residuals
- `glance()`: a one-row model summary

All of these are much friendlier than a fitted `lm()` object and set us up for interesting downstream analyses and plots.

12.3.8 Simplify and Combine

Let's look at the result of `broom::tidy()` for a single model. Walk before you run.

```
library(broom)
tidy(gap_nested$fit[[1]])
##      term estimate std.error statistic    p.value
## 1 (Intercept) 29.3566375 0.69898128  41.99918 1.404235e-12
## 2 I(year - 1950) 0.2753287 0.02045093  13.46289 9.835213e-08
```

We get a two-row data frame, one with results for the intercept and one for the slope. This is much more approachable than fitted `lm` objects!

Apply `tidy()` to the model for each country with the same `purrr::map()` inside `mutate()` strategy as above.

```
(gap_nested <- gap_nested %>%
  mutate(tidy = purrr::map(fit, tidy)))
## # A tibble: 142 x 5
##   continent country      data fit      tidy
##   <fctr>    <fctr>    <list> <list>    <list>
## 1 Asia Afghanistan <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 2 Europe  Albania <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 3 Africa  Algeria <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 4 Africa  Angola <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 5 Americas Argentina <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 6 Oceania Australia <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 7 Europe  Austria <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 8 Asia    Bahrain <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## 9 Asia    Bangladesh <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
```

```
## 10 Europe Belgium <tibble [12 x 4]> <S3: lm> <data.frame [2 x 5]>
## # ... with 132 more rows
```

The last step is now to simplify, preferably back to a normal tibble. We do this by retaining variables that are amenable to simplification and using `unnest()`, thus completing the circle.

```
(gap_coefs <- gap_nested %>%
  select(continent, country, tidy) %>%
  unnest(tidy))
## # A tibble: 284 x 7
##   continent country      term estimate std.error statistic
##   <fctr>    <fctr>    <chr>    <dbl>    <dbl>    <dbl>
## 1 Asia Afghanistan (Intercept) 29.3566375 0.698981278 41.999176
## 2 Asia Afghanistan I(year - 1950) 0.2753287 0.020450934 13.462890
## 3 Europe Albania (Intercept) 58.5597618 1.133575812 51.659325
## 4 Europe Albania I(year - 1950) 0.3346832 0.033166387 10.091036
## 5 Africa Algeria (Intercept) 42.2364149 0.756269040 55.848399
## 6 Africa Algeria I(year - 1950) 0.5692797 0.022127070 25.727749
## 7 Africa Angola (Intercept) 31.7079741 0.804287463 39.423683
## 8 Africa Angola I(year - 1950) 0.2093399 0.023532003 8.895964
## 9 Americas Argentina (Intercept) 62.2250191 0.167091314 372.401279
## 10 Americas Argentina I(year - 1950) 0.2317084 0.004888791 47.395847
## # ... with 274 more rows, and 1 more variables: p.value <dbl>
```

12.3.9 Recap

The whole point of this was to apply a computation to all the pieces of a dataset and glue the results back together. First, let's review all of our work so far in one place. It's remarkably compact.

```
gap_nested <- gapminder %>%
  group_by(continent, country) %>%
  nest()

le_vs_yr <- function(df) {
  lm(lifeExp ~ I(year - 1950), data = df)
}

gap_coefs <- gap_nested %>%
  mutate(fit = purrr::map(data, le_vs_yr),
         tidy = purrr::map(fit, tidy)) %>%
  select(continent, country, tidy) %>%
  unnest(tidy)
```

Reflect on how you would have obtained a data frame of country-specific intercepts and slopes from a regression of life expectancy on year. Did you know any approaches to solving this

problem? If no, then rejoice that you now have one! If yes, does the approach outlined here seem simpler?

List-columns take some getting used to, but they are a required component of the strategy laid out above for general-purpose split-apply-combine.

12.3.10 Enjoy the Payoff of Your Work

Let's celebrate by exploring the estimated slopes and intercepts a bit. First, we recode the variables corresponding to "intercept" and "slope."

```
(gap_coefs <- gap_coefs %>%  
  mutate(term = recode(term,  
    `(Intercept)` = "intercept",  
    `I(year - 1950)` = "slope"))  
## # A tibble: 284 x 7  
##   continent country term estimate std.error statistic  
##   <fctr>    <fctr> <chr>    <dbl>    <dbl>    <dbl>  
## 1 Asia Afghanistan intercept 29.3566375 0.698981278 41.999176  
## 2 Asia Afghanistan slope 0.2753287 0.020450934 13.462890  
## 3 Europe Albania intercept 58.5597618 1.133575812 51.659325  
## 4 Europe Albania slope 0.3346832 0.033166387 10.091036  
## 5 Africa Algeria intercept 42.2364149 0.756269040 55.848399  
## 6 Africa Algeria slope 0.5692797 0.022127070 25.727749  
## 7 Africa Angola intercept 31.7079741 0.804287463 39.423683  
## 8 Africa Angola slope 0.2093399 0.023532003 8.895964  
## 9 Americas Argentina intercept 62.2250191 0.167091314 372.401279  
## 10 Americas Argentina slope 0.2317084 0.004888791 47.395847  
## # ... with 274 more rows, and 1 more variables: p.value <dbl>
```

Due to the way we parametrized the model, the intercepts correspond to expected life expectancy in 1950. These numbers should be plausible as human life expectancies. The slopes correspond to change in expected life expectancy from one year to the next. We expect positive numbers to dominate, but they'll probably be less than one. A reshaped version of the estimates, `gap_ests`, is handy for numerical summarization and visualization.

```
(gap_ests <- gap_coefs %>%  
  select(continent:estimate) %>%  
  spread(key = term, value = estimate))  
## # A tibble: 142 x 4  
##   continent country intercept slope  
## * <fctr>    <fctr>    <dbl>    <dbl>  
## 1 Africa    Algeria 42.23641 0.56927972  
## 2 Africa    Angola 31.70797 0.20933986  
## 3 Africa    Benin 38.92005 0.33423287  
## 4 Africa    Botswana 52.80778 0.06066853  
## 5 Africa    Burkina Faso 33.95674 0.36397483
```

```
## 6 Africa Burundi 40.27037 0.15413427
## 7 Africa Cameroon 40.74917 0.25014685
## 8 Africa Central African Republic 38.44170 0.18390559
## 9 Africa Chad 39.30288 0.25324406
## 10 Africa Comoros 39.09522 0.45039091
## # ... with 132 more rows
```

```
gap_estimates %>%
  select(intercept, slope) %>%
  summary()
## intercept slope
## Min. :27.24 Min. :-0.09302
## 1st Qu.:39.36 1st Qu.: 0.20832
## Median :47.42 Median : 0.32145
## Mean :49.86 Mean : 0.32590
## 3rd Qu.:62.05 3rd Qu.: 0.44948
## Max. :71.95 Max. : 0.77218
```

The numerical summaries look reasonable. We conclude with a look at the full distribution.

```
ggplot(gap_coefs, aes(x = estimate)) +
  geom_density() + geom_rug() + facet_wrap(~ term, scales = "free")
ggplot(gap_estimates, aes(x = intercept, y = slope)) +
  geom_point() +
  geom_smooth(se = FALSE, lwd = 2)
```

Figure 12.1 Distributions of model intercepts and slopes and their relationship in a scatter plot

12.4 EXERCISE

1. Fit linear regression models of the daily bike counts on precipitation and max temperature, first for both bridges together and then for each bridge separately using the split-apply-combine pattern.
2. When using ggfortify to plot weekly variation, trend and noise separately, you need to plot each bridge separately (sample code [here](#)). Use the split-apply-combine to avoid having to repeat for each bridge.

12.5 RESOURCES:

1. [purrr package](#)
2. [purrr tutorial](#)
3. [Software Carpentry lesson on Split-Apply-Combine](#)

13.0 WHERE TO GO FROM HERE

Practice makes perfection. Use what you learned here as much as possible in your own research and work, even though you may not master everything first off. Once you find a solution to your data science problem, think about whether it conforms to the [best practices for data science](#). If not, can you improve your solution according to the best practices?

13.1 NOTICEABLE FEATURES NOT COVERED

R is getting more powerful all the time. There are many exciting features and packages that we have not covered and much more are emerging. Below are a few features I think are worth mentioning.

13.1.1 Automate Testing Your Code

It is always a good idea to test your code with simple inputs to make sure it works correctly before adding complexity and deploying it for actual work (see [best practices for data science](#) 5). There are ways to automate the tests so you will not forget to run them. Hadley Wickham writes about [testing for R packages](#) and [testthat is an R package specifically to make testing](#). And if you use GitHub or other popular public version control websites, it is possible to use Travis-CI to do "continuous integration" tests for your code. Julia Silge has [a nice tutorial for beginners](#).

13.1.2 Automate Your Workflow

If you are a seasoned programmer and are familiar with the make toolchain, you can use it to automate your workflow too. Professor Jenny Bryan provides [a tutorial](#) for her STAT 545 class. A more research-oriented solution is to use [the remake package](#):

The idea here is to re-imagine a set of ideas from make but built for R. Rather than having a series of calls to different instances of R (as happens if you run make on R scripts), the idea is to define pieces of a pipeline within an R session. Rather than being language agnostic (like make must be), remake is unapologetically R focused.

remake is still under heavy development and is not yet available on CRAN. You will have to install it from GitHub:

```
library(devtools)
install_github("richfitz/remake")
```

13.1.3 Develop Interactive Apps With Shiny

Shiny allows you to develop interactive apps that run on top of R. Again, Professor Jenny Bryan has a nice [tutorial for getting started with Shiny](#).

13.2 WHERE TO FIND HELP

1. Help documents;
2. Package vignettes and manual;
3. [R cookbook](#);
4. [Stack Overflow](#) and [Cross Validated](#), A Q&A community for programming and statistics;
5. Google is your friend; and
6. Ask a person who may know.

REFERENCES

- Merali, Zeeya. 2010. “Computational Science: ...Error.” *Nature News* 467 (7317): 775–77. doi:[10.1038/467775a](https://doi.org/10.1038/467775a).
- University Transportation Research Center. 2014. “Ground Transportation Technology Symposium: Big Data and Innovative Solutions for Safe, Efficient and Sustainable Mobility.” In. New York, NY.
- Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. “Best Practices for Scientific Computing.” *PLOS Biology* 12 (1): e1001745. doi:[10.1371/journal.pbio.1001745](https://doi.org/10.1371/journal.pbio.1001745).

Transportation Research and Education Center
Portland State University
1900 S.W. Fourth Ave., Suite 175
Portland, OR 97201