Programming Skills for Data Science
The Pearson Addison-Wesley Data and Analytics Series provides readers with practical knowledge for solving problems and answering questions with data. Titles in this series primarily focus on three areas:

1. **Infrastructure**: how to store, move, and manage data
2. **Algorithms**: how to mine intelligence or make predictions based on data
3. **Visualizations**: how to represent data and insights in a meaningful and compelling way

The series aims to tie all three of these areas together to help the reader build end-to-end systems for fighting spam; making recommendations; building personalization; detecting trends, patterns, or problems; and gaining insight from the data exhaust of systems and user interactions.

Visit informit.com/awdataseries for a complete list of available publications.

Make sure to connect with us!
informit.com/socialconnect
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2018953978

Copyright © 2019 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-10: 0-13-513310-6
To our students who challenged us to develop better resources, and our families who supported us in the process.
Contents

Foreword xi
Preface xiii
Acknowledgments xvii
About the Authors xix

I: Getting Started 1

1 Setting Up Your Computer 3
1.1 Setting up Command Line Tools 4
1.2 Installing git 5
1.3 Creating a GitHub Account 6
1.4 Selecting a Text Editor 6
1.5 Downloading the R Language 7
1.6 Downloading RStudio 8

2 Using the Command Line 9
2.1 Accessing the Command Line 9
2.2 Navigating the File System 11
2.3 Managing Files 15
2.4 Dealing with Errors 18
2.5 Directing Output 20
2.6 Networking Commands 20

II: Managing Projects 25

3 Version Control with git and GitHub 27
3.1 What is git? 27
3.2 Configuration and Project Setup 30
3.3 Tracking Project Changes 32
3.4 Storing Projects on GitHub 36
3.5 Accessing Project History 40
3.6 Ignoring Files from a Project 42

4 Using Markdown for Documentation 45
4.1 Writing Markdown 45
4.2 Rendering Markdown 48
III: Foundational R Skills  51

5 Introduction to R  53
  5.1 Programming with R  53
  5.2 Running R Code  54
  5.3 Including Comments  58
  5.4 Defining Variables  58
  5.5 Getting Help  63

6 Functions  69
  6.1 What Is a Function?  69
  6.2 Built-in R Functions  71
  6.3 Loading Functions  73
  6.4 Writing Functions  75
  6.5 Using Conditional Statements  79

7 Vectors  81
  7.1 What Is a Vector?  81
  7.2 Vectorized Operations  83
  7.3 Vector Indices  88
  7.4 Vector Filtering  90
  7.5 Modifying Vectors  92

8 Lists  95
  8.1 What Is a List?  95
  8.2 Creating Lists  96
  8.3 Accessing List Elements  97
  8.4 Modifying Lists  100
  8.5 Applying Functions to Lists with lapply()  102

IV: Data Wrangling  105

9 Understanding Data  107
  9.1 The Data Generation Process  107
  9.2 Finding Data  108
  9.3 Types of Data  110
  9.4 Interpreting Data  112
  9.5 Using Data to Answer Questions  116
Foreword

The data science skill set is ever-expanding to include more and more of the analytics pipeline. In addition to fitting statistical and machine learning models, data scientists are expected to ingest data from different file formats, interact with APIs, work at the command line, manipulate data, create plots, build dashboards, and track all their work in git. By combining all of these components, data scientists can produce amazing results. In this text, Michael Freeman and Joel Ross have created *the* definitive resource for new and aspiring data scientists to learn foundational programming skills.

Michael and Joel are best known for leveraging visualization and front-end interfaces to compose explanations of complex data science topics. In addition to their written work, they have created interactive explanations of statistical methods, including a particularly clarifying and captivating introduction to hierarchical modeling. It is this sensibility and deep commitment to demystifying complicated topics that they bring to their new book, which teaches a plethora of data science skills.

This tour of data science begins by setting up the local computing environment such as text editors, RStudio, the command line, and git. This lays a solid foundation—that is far too often glossed over—making it easier to learn core data skills. After this, those core skills are given attention, including data manipulation, visualization, reporting, and an excellent explanation of APIs. They even show how to use git collaboratively, something data scientists all too often neglect to integrate into their projects.

*Programming Skills for Data Science* lives up to its name in teaching the foundational skills needed to get started in data science. This book provides valuable insights for both beginners and those with more experience who may be missing some key knowledge. Michael and Joel made full use of their years of teaching experience to craft an engrossing tutorial.

—Jared Lander, series editor
This page intentionally left blank
Preface

Transforming data into actionable information requires the ability to clearly and reproducibly wrangle, analyze, and visualize that data. These skills are the foundations of data science, a field that has amplified our collective understanding of issues ranging from disease transmission to racial inequities. Moreover, the ability to programmatically interact with data enables researchers and professionals to quickly discover and communicate patterns in data that are often difficult to detect. Understanding how to write code to work with data allows people to engage with information in new ways and on larger scales.

The existence of free and open source software has made these tools accessible to anyone with access to a computer. The purpose of this book is to teach people how to leverage programming to ask questions of their data sets.

Focus of the Book

This book revolves around the practical steps needed to program for data science using the R programming language. It takes a holistic approach to teaching the topic, recognizing that an entire ecosystem of tools and technologies is needed to do this. While writing code is a core part of being a data scientist (and this book), many more foundational skills must be acquired as part of this journey. Data science requires installing and configuring software to write, execute, and manage code; tracking the version of (and changes to) your projects; leveraging core concepts from computer science to understand how to accomplish a given task; accessing and processing data from a variety of sources; leveraging visual communication to expose patterns in your data; and building applications to share insights with others. The purpose of this text is to help people develop a strong foundation across these areas so that they can enter the data science field (or bring data science to their field).

Who Should Read This Book

This book is written for people with no programming or data science experience, though it would still be helpful for people active in the field. This book was originally developed to support a course in the Informatics undergraduate degree program at the University of Washington, so it is (not surprisingly) well suited for college students interested in entering the data science field. We also believe that anyone whose job involves working with data can benefit from learning how to reproducibly create analyses, visualizations, and reports.

If you are interested in pursuing a career in data science, or if you use data on a regular basis and want to use programming techniques to gain information from that data, then this text is for you.
Book Structure

The book is divided into six sections, each of which is summarized here.

Part I: Getting Started
This section walks through the steps of downloading and installing necessary software for the rest of the book. More specifically, Chapter 1 details how to install a text editor, Bash terminal, the R interpreter, and the RStudio program. Then, Chapter 2 describes how to use the command line for basic file system navigation.

Part II: Managing Projects
This section walks through the technical basis of project management, including keeping track of the version of your code and producing documentation. Chapter 3 introduces the git software to track line-by-line code changes, as well as the corresponding popular code hosting and collaboration service GitHub. Chapter 4 then describes how to use Markdown to produce the well-structured and -styled documentation needed for sharing and presenting data.

Part III: Foundational R Skills
This section introduces the R programming language, the primary language used throughout the book. In doing so, it introduces the basic syntax of the language (Chapter 5), describes fundamental programming concepts such as functions (Chapter 6), and introduces the basic data structures of the language: vectors (Chapter 7), and lists (Chapter 8).

Part IV: Data Wrangling
Because the most time-consuming part of data science is often loading, formatting, exploring, and reshaping data, this section of the book provides a deep dive into the best ways to wrangle data in R. After introducing techniques and concepts for understanding the structure of real-world data (Chapter 9), the book presents the data structure most commonly used for managing data in R: the data frame (Chapter 10). To better support working with this data, the book then describes two packages for programmatically interacting with the data: dplyr (Chapter 11), and tidyr (Chapter 12). The last two chapters of the section describe how to load data from databases (Chapter 13) and web-based data services with application programming interfaces (APIs) (Chapter 14).

Part V: Data Visualization
This section of the book focuses on the conceptual and technical skills necessary to design and build visualizations as part of the data science process. It begins with an overview of data visualization principles (Chapter 15) to guide your choices in designing visualizations. Chapter 16 then describes in granular detail how to use the ggplot2 visualization package in R. Finally, Chapter 17 explores the use of three additional R packages for producing engaging interactive visualizations.

Part VI: Building and Sharing Applications
As in any domain, data science insights are valuable only if they can be shared with and understood by others. The final section of the book focuses on using two different approaches to creating interactive platforms to share your insights (directly from your R program!). Chapter 18 uses the R
Markdown framework to transform analyses into sharable documents and websites. Chapter 19 takes this a step further with the Shiny framework, which allows you to create interactive web applications using R. Chapter 20 then describes approaches for working on collaborative teams of data scientists, and Chapter 21 details how you can further your education beyond this book.

Book Conventions

Throughout the book, you will see computer code appear inline with the text, as well as in distinct blocks. When code appears inline, it will appear in monospace font. A distinct code block looks like this:

```r
# This is a comment - it describes the code that follows
# The next line of code prints the text "Hello world!"
print("Hello world!")
```

The text in the code blocks is colored to reflect the syntax of the programming language used (typically the R language). Example code blocks often include values that you need to replace. These replacement values appear in UPPER_CASE_FONT, with words separated by underscores. For example, if you need to work with a folder of your choosing, you would put the name of your folder where it says FOLDER_NAME in the code. Code sections will all include comments: in programming, comments are bits of text that are not interpreted as computer instructions—they aren’t code, they’re just notes about the code! While a computer is able to understand the code, comments are there to help people understand it. Tips for writing your own descriptive comments are discussed in Chapter 5.

To guide your reading, we also include five types of special callout notes:

- **Tip**: These boxes provide best practices and shortcuts that can make your life easier.
- **Fun Fact**: These boxes provide interesting background information on a topic.
- **Remember**: These boxes reinforce key points that are important to keep in mind.
- **Caution**: These boxes describe common mistakes and explain how to avoid them.
- **Going Further**: These boxes suggest resources for expanding your knowledge beyond this text.

Throughout the text there are instructions for using specific keyboard keys. These are included in the text in lowercase monospace font. When multiple keys need to be pressed at the same time, they are separated by a plus sign (+). For example, if you needed to press the Command and “c” keys at the same time, it would appear as Cmd+c.

Whenever the cmd key is used, Windows users should instead use the Control (ctrl) key.
How to Read This Book

The individual chapters in this book will walk you through the process of programming for data science. Chapters often build upon earlier examples and concepts (particularly through Part III and Part IV).

This book includes a large number of code examples and demonstrations, with reported output and results. That said, the best way to learn to program is to do it, so we highly recommend that as you read, you type out the code examples and try them yourself! Experiment with different options and variations—if you’re wondering how something works or if an option is supported, the best thing to do is try it yourself. This will help you not only practice the actual writing of code, but also better develop your own mental model of how data science programs work.

Many chapters conclude by applying the described techniques to a real data set in an In Action section. These sections take a data-driven approach to understanding issues such as gentrification, investment in education, and variation in life expectancy around the world. These sections use a hands-on approach to using new skills, and all code is available online.¹

As you move through each chapter, you may want to complete the accompanying set of online exercises.² This will help you practice new techniques and ensure your understanding of the material. Solutions to the exercises are also available online.

Finally, you should know that this text does not aim to be comprehensive. It is both impractical and detrimental to learning to attempt to explain every nuance and option in the R language and ecosystem (particularly to people who are just starting out). While we discuss a large number of popular tools and packages, the book cannot explain all possible options that exist now or will be created in the future. Instead, this text aims to provide a primer on each topic—giving you enough details to understand the basics and to get up and running with a particular data science programming task. Beyond those basics, we provide copious links and references to further resources where you can explore more and dive deeper into topics that are relevant or of interest to you. This book will provide the foundations of using R for data science—it is up to each reader to apply and build upon those skills.

Accompanying Code

To guide your learning, a set of online exercises (and their solutions) is available for each chapter. The complete analysis code for all seven In Action sections is also provided. See the book website³ for details.

¹In-Action Code: https://github.com/programming-for-data-science/in-action
²Book Exercises: https://github.com/programming-for-data-science
³https://programming-for-data-science.github.io
Acknowledgments

We would like to thank the University of Washington Information School for providing us with an environment in which to collaborate and develop these materials. We had the support of many faculty members—in particular, David Stearns (who contributed to the materials on version control) as well as Jessica Hullman and Ott Toomet (who provided initial feedback on the text). We also thank Kevin Hodges, Jason Baik, and Jared Lander for their comments and insights, as well as Debra Williams Cauley, Julie Nahil, Rachel Paul, Jill Hobbs, and the staff at Pearson for their work bringing this book to press.

Finally, this book would not have been possible without the extraordinary open source community around the R programming language.
This page intentionally left blank
About the Authors

**Michael Freeman** is a Senior Lecturer at the University of Washington Information School, where he teaches courses in data science, interactive data visualization, and web development. Prior to his teaching career, he worked as a data visualization specialist and research fellow at the Institute for Health Metrics and Evaluation. There, he performed quantitative global health research and built a variety of interactive visualization systems to help researchers and the public explore global health trends.

Michael is interested in applications of data science to social justice, and holds a Master’s in Public Health from the University of Washington. (His faculty page is at https://faculty.washington.edu/mikefree/.)

**Joel Ross** is a Senior Lecturer at the University of Washington Information School, where he teaches courses in web development, mobile application development, software architecture, and introductory programming. While his primary focus is on teaching, his research interests include games and gamification, pervasive systems, computer science education, and social computing. He has also done research on crowdsourcing systems, human computation, and encouraging environmental sustainability.

Joel earned his M.S. and Ph.D. in Information and Computer Sciences from the University of California, Irvine. (His faculty page is at https://faculty.washington.edu/joelross/.)
Accessing Web APIs

Previous chapters have described how to access data from local .csv files, as well as from local databases. While working with local data is common for many analyses, more complex shared data systems leverage web services for data access. Rather than store data on each analyst’s computer, data is stored on a remote server (i.e., a central computer somewhere on the internet) and accessed similarly to how you access information on the web (via a URL). This allows scripts to always work with the latest data available when performing analysis of data that may be changing rapidly, such as social media data.

In this chapter, you will learn how to use R to programmatically interact with data stored by web services. From an R script, you can read, write, and delete data stored by these services (though this book focuses on the skill of reading data). Web services may make their data accessible to computer programs like R scripts by offering an application programming interface (API). A web service’s API specifies where and how particular data may be accessed, and many web services follow a particular style known as REpresentational State Transfer (REST).¹ This chapter covers how to access and work with data from these RESTful APIs.

14.1 What Is a Web API?

An interface is the point at which two different systems meet and communicate, exchanging information and instructions. An application programming interface (API) thus represents a way of communicating with a computer application by writing a computer program (a set of formal instructions understandable by a machine). APIs commonly take the form of functions that can be called to give instructions to programs. For example, the set of functions provided by a package like dplyr make up the API for that package.

While some APIs provide an interface for leveraging some functionality, other APIs provide an interface for accessing data. One of the most common sources of these data APIs are web services—that is, websites that offer an interface for accessing their data.

With web services, the interface (the set of “functions” you can call to access the data) takes the form of HTTP requests—that is, requests for data sent following the HyperText Transfer Protocol.

This is the same protocol (way of communicating) used by your browser to view a webpage! An HTTP request represents a message that your computer sends to a web server: another computer on the internet that “serves,” or provides, information. That server, upon receiving the request, will determine what data to include in the response it sends back to the requesting computer. With a web browser, the response data takes the form of HTML files that the browser can render as webpages. With data APIs, the response data will be structured data that you can convert into R data types such as lists or data frames.

In short, loading data from a web API involves sending an HTTP request to a server for a particular piece of data, and then receiving and parsing the response to that request.

Learning how to use web APIs will greatly expand the available data sets you may want to use for analysis. Companies and services with large amounts of data, such as Twitter, iTunes, or Reddit, make (some of) their data publicly accessible through an API. This chapter will use the GitHub API to demonstrate how to work with data stored in a web service.

14.2 RESTful Requests

There are two parts to a request sent to a web API: the name of the resource (data) that you wish to access, and a verb indicating what you want to do with that resource. In a way, the verb is the function you want to call on the API, and the resource is an argument to that function.

14.2.1 URIs

Which resource you want to access is specified with a Uniform Resource Identifier (URI). A URI is a generalization of a URL (Uniform Resource Locator)—what you commonly think of as a “web address.” A URI acts a lot like the address on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department and the person to receive the letter, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource, illustrated in Figure 14.1.

![Figure 14.1 The format (schema) of a URI.](https://domain.com:9999/example/page/type=husky&name=dubs#nose)

---

2 Twitter API: https://developer.twitter.com/en/docs
4 Reddit API: https://www.reddit.com/dev/api/
5 GitHub API: https://developer.github.com/v3/
Not all parts of the URI are required. For example, you don’t necessarily need a port, query, or fragment. Important parts of the URI include:

- **scheme (protocol):** The “language” that the computer will use to communicate the request to the API. With web services this is normally https (secure HTTP).
- **domain:** The address of the web server to request information from.
- **path:** The identifier of the resource on that web server you wish to access. This may be the name of a file with an extension if you’re trying to access a particular file, but with web services it often just looks like a folder path!
- **query:** Extra parameters (arguments) with further details about the resource to access.

The domain and path usually specify the location of the resource of interest. For example, www.domain.com/users might be an *identifier for a resource* that serves information about all the users. Web services can also have “subresources” that you can access by adding extra pieces to the path. For example, www.domain.com/users/layla might access the specific resource (“layla”) that you are interested in.

With web APIs, the URI is often viewed as being broken up into three parts, as shown in Figure 14.2:

- The **base URI** is the domain that is included on *all* resources. It acts as the “root” for any particular endpoint. For example, the GitHub API has a base URI of https://api.github.com. All requests to the GitHub API will have that base.
- An **endpoint** is the location that holds the specific information you want to access. Each API will have many different endpoints at which you can access specific data resources. The GitHub API, for example, has different endpoints for /users and /orgs so that you can access data about users or organizations, respectively.

Note that many endpoints support accessing multiple subresources. For example, you can access information about a specific user at the endpoint /users/:username. The colon : indicates that the subresource name is a *variable*—you can replace that part of the endpoint with whatever string you want. Thus if you were interested in the GitHub user nbremer, you would access the /users/nbremer endpoint.

Subresources may have further subresources (which may or may not have variable names). The endpoint /orgs/:org/repos refers to the list of repositories belonging to an organization. Variable names in endpoints might alternatively be written inside of curly braces {}—for example, /orgs/{org}/repos. Neither the colon nor the braces are

![Figure 14.2](https://api.github.com/search/repositories?q=dplyr&sort=forks)

**Figure 14.2** The anatomy of a web API request URI.

---

7Nadieh Bremer, freelance data visualization designer: https://www.visualcinnamon.com
programming language syntax; instead, they are common conventions used to communicate how to specify endpoints.

- **Query parameters** allow you to specify additional information about which exact information you want from the endpoint, or how you want it to be organized (see Section 14.2.1.1 for more details).

**Remember:** One of the biggest challenges in accessing a web API is understanding what resources (data) the web service makes available and which endpoints (URIs) can request those resources. Read the web service's documentation carefully—popular services often include examples of URIs and the data returned from them.

A query is constructed by appending the endpoint and any query parameters to the base URI. For example, so you could access a GitHub user by combining the base URI (https://api.github.com) and endpoint (/users/nbremer) into a single string: https://api.github.com/users/nbremer. Sending a request to that URI will return data about the user—you can send this request from an R program or by visiting that URI in a web browser, as shown in Figure 14.3. In short, you can access a particular data resource by sending a request to a particular endpoint.

Indeed, one of the easiest ways to make a request to a web API is by navigating to the URI using your web browser. Viewing the information in your browser is a great way to explore the resulting data, and make sure you are requesting information from the proper URI (i.e., that you haven’t made a typo in the URI).

**Tip:** The JSON format (see Section 14.4) of data returned from web APIs can be quite messy when viewed in a web browser. Installing a browser extension such as JSONView will format the data in a somewhat more readable way. Figure 14.3 shows data formatted with this extension.

14.2.1.1 Query Parameters

Web URIs can optionally include **query parameters**, which are used to request a more specific subset of data. You can think of them as additional optional arguments that are given to the request function—for example, a keyword to search for or criteria to order results by.

The query parameters are listed at the end of a URI, following a question mark (?) and are formed as key–value pairs similar to how you named items in lists. The **key** (parameter name) is listed first, followed by an equals sign (=), followed by the **value** (parameter value), with no spaces between anything. You can include multiple query parameters by putting an ampersand (&) between each key–value pair. You can see an example of this syntax by looking at the URL bar in a web browser when you use a search engine such as Google or Yahoo, as shown in Figure 14.4. Search engines produce URLs with a lot of query parameters, not all of which are obvious or understandable.
14.2 RESTful Requests

Notice that the exact query parameter name used differs depending on the web service. Google uses a `q` parameter (likely for “query”) to store the search term, while Yahoo uses a `p` parameter.

Similar to arguments for functions, API endpoints may either require query parameters (e.g., you must provide a search term) or optionally allow them (e.g., you may provide a sorting order). For example, the GitHub API has a `/search/repositories` endpoint that allows users to search for a specific repository: you are required to provide a `q` parameter for the query, and can optionally provide a `sort` parameter for how to sort the results:

```
# A GitHub API URI with query parameters: search term `q` and sort
# order `sort`
https://api.github.com/search/repositories?q=dplyr&sort=forks
```
Chapter 14 Accessing Web APIs

Figure 14.4 Search engine URLs for Google (top) and Yahoo (bottom) with query parameters (underlined in blue). The “search term” parameter for each web service is underlined in red.

Results from this request are shown in Figure 14.5.

Caution: Many special characters (e.g., punctuation) cannot be included in a URL. This group includes characters such as spaces! Browsers and many HTTP request packages will automatically encode these special characters into a usable format (for example, converting a space into a %20), but sometimes you may need to do this conversion yourself.

14.2.1.2 Access Tokens and API Keys

Many web services require you to register with them to send them requests. This allows the web service to limit access to the data, as well as to keep track of who is asking for which data (usually so that if someone starts “spamming” the service, that user can be blocked).

To facilitate this tracking, many services provide users with access tokens (also called API keys). These unique strings of letters and numbers identify a particular developer (like a secret password that works just for you). Furthermore, your API key can provide you with additional access to information based on which user you are. For example, when you get an access key for the GitHub API, that key will provide you with additional access and control over your repositories. This enables you to request information about private repos, and even programmatically interact with GitHub through the API (i.e., you can delete a repo—so tread carefully!).

Web services will require you to include your access token in the request, usually as a query parameter; the exact name of the parameter varies, but it often looks like access_token or api_key. When exploring a web service, keep an eye out for whether it requires such tokens.

---

8GitHub API, delete a repository https://developer.github.com/v3/repos/#delete-a-repository
14.2 RESTful Requests

Figure 14.5  A subset of the GitHub API response returned by the URI https://api.github.com/search/repositories?q=dplyr&sort=forks, as displayed in a web browser.

**Caution:** Watch out for APIs that mention using an authentication service called OAuth when explaining required API keys. OAuth is a system for performing authentication—that is, having someone prove that they are who they say they are. OAuth is generally used to let someone log into a website from your application (like what a “Log in with Google” button does). OAuth systems require more than one access key, and these keys must be kept secret. Moreover, they usually require you to run a web server to use them correctly (which requires significant extra setup; see the full httr documentation\(^*\) for details). You can do this in R, but may want to avoid this challenge while learning how to use APIs.

\(^*\)https://cran.r-project.org/web/packages/httr/httr.pdf
Access tokens are a lot like passwords; you will want to keep them secret and not share them with others. This means that you should not include them in any files you commit to git and push to GitHub. The best way to ensure the secrecy of access tokens in R is to create a separate script file in your repo (e.g., api_keys.R) that includes exactly one line, assigning the key to a variable:

```r
# Store your API key from a web service in a variable
# It should be in a separate file (e.g., `api_keys.R`)
api_key <- "123456789abcdefg"
```

To access this variable in your “main” script, you can use the `source()` function to load and run your api_keys.R script (similar to clicking the Source button to run a script). This function will execute all lines of code in the specified script file, as if you had “copy-and-pasted” its contents and run them all with `ctrl+enter`. When you use `source()` to execute the api_keys.R script, it will execute the code statement that defines the api_key variable, making it available in your environment for your use:

```r
# In your “main” script, load your API key from another file
# (Make sure working directory is set before running the following code!)
source("api_keys.R")  # load the script using a *relative path*
print(api_key)  # the key is now available!
```

Anyone else who runs the script will need to provide an api_key variable to access the API using that user’s own key. This practice keeps everyone’s account separate.

You can keep your api_keys.R file from being committed by including the filename in the `.gitignore` file in your repo; that will keep it from even possibly being committed with your code! See Chapter 3 for details about working with the `.gitignore` file.

### 14.2.2 HTTP Verbs

When you send a request to a particular resource, you need to indicate what you want to do with that resource. This is achieved by specifying an **HTTP verb** in the request. The HTTP protocol supports the following verbs:

- **GET**: Return a representation of the current state of the resource.
- **POST**: Add a new subresource (e.g., insert a record).
- **PUT**: Update the resource to have a new state.
- **PATCH**: Update a portion of the resource’s state.
- **DELETE**: Remove the resource.
- **OPTIONS**: Return the set of methods that can be performed on the resource.
By far the most commonly used verb is **GET**, which is used to “get” (download) data from a web service—this is the type of request that is sent when you enter a URL into a web browser. Thus you would send a GET request for the `/users/nbremer` endpoint to access that data resource.

Taken together, this structure of treating each datum on the web as a resource that you can interact with via HTTP requests is referred to as the **REST architecture** (**RE**presentational **ST**ate **T**ransfer). Thus, a web service that enables data access through named resources and responds to HTTP requests is known as a **RESTful** service, that has a RESTful API.

### 14.3 Accessing Web APIs from R

To access a web API, you just need to send an HTTP request to a particular URI. As mentioned earlier, you can easily do this with the browser: navigate to a particular address (base URI + endpoint), and that will cause the browser to send a GET request and display the resulting data. For example, you can send a request to the GitHub API to search for repositories that match the string “dplyr” (see the response in Figure 14.5):

```r
# The URI for the `search/repositories` endpoint of the GitHub API: query # for `dplyr`, sorting by `forks`
https://api.github.com/search/repositories?q=dplyr&sort=forks
```

This query accesses the `/search/repositories` endpoint, and also specifies two query parameters:

- **q**: The term(s) you are searching for
- **sort**: The attribute of each repository that you would like to use to sort the results (in this case, the number of forks of the repo)

(Note that the data you will get back is structured in JSON format. See Section 14.4 for details.)

While you can access this information using your browser, you will want to load it into **R** for analysis. In **R**, you can send GET requests using the **httr** package. As with **dplyr**, you will need to install and load this package to use it:

```r
install.packages("httr") # once per machine
library("httr") # in each relevant script
```

This package provides a number of functions that reflect HTTP verbs. For example, the **GET()** function will send an HTTP GET request to the URI:

```r
# Make a GET request to the GitHub API's `/search/repositories` endpoint # Request repositories that match the search "dplyr", and sort the results # by forks
url <- "https://api.github.com/search/repositories?q=dplyr&sort=forks"
response <- GET(url)
```

This code will make the same request as your web browser, and store the response in a variable called `response`. While it is possible to include query parameters in the URI string (as above), **httr**

---

*Getting started with http: official quickstart guide for http: https://cran.r-project.org/web/packages/httr/vignettes/quickstart.html*
also allows you to include them as a list passed as a query argument. Furthermore, if you plan on accessing multiple different endpoints (which is common), you can structure your code a bit more modularly, as described in the following example; this structure makes it easy to set and change variables (instead of needing to do a complex `paste()` operation to produce the correct string):

```r
# Restructure the previous request to make it easier to read and update. DO THIS.

# Make a GET request to the GitHub API's "search/repositories" endpoint
# Request repositories that match the search "dplyr", sorted by forks

# Construct your `resource_uri` from a reusable `base_uri` and an `endpoint`
base_uri <- "https://api.github.com"
endpoint <- "/search/repositories"
resource_uri <- paste0(base_uri, endpoint)

# Store any query parameters you want to use in a list
query_params <- list(q = "dplyr", sort = "forks")

# Make your request, specifying the query parameters via the `query` argument
response <- GET(resource_uri, query = query_params)
```

If you try printing out the response variable that is returned by the `GET()` function, you will first see information about the response:

```
Response [https://api.github.com/search/repositories?q=dplyr&sort=forks]
  Date: 2018-03-14 06:43
  Status: 200
  Content-Type: application/json; charset=utf-8
  Size: 171 kB
```

This is called the response header. Each response has two parts: the header and the body. You can think of the response as an envelope: the header contains meta-data like the address and postage date, while the body contains the actual contents of the letter (the data).

**Tip:** The URI shown when you print out the response variable is a good way to check exactly which URI you sent the request to: copy that into your browser to make sure it goes where you expected!

Since you are almost always interested in working with the response body, you will need to extract that data from the response (e.g., open up the envelope and pull out the letter). You can do this with the `content()` function:

```r
# Extract content from `response`, as a text string
response_text <- content(response, type = "text")
```

Note the second argument `type = "text"`; this is needed to keep `httr` from doing its own processing on the response data (you will use other methods to handle that processing).
14.4 Processing JSON Data

Now that you’re able to load data into R from an API and extract the content as text, you will need to transform the information into a usable format. Most APIs will return data in JavaScript Object Notation (JSON) format. Like CSV, JSON is a format for writing down structured data—but, while .csv files organize data into rows and columns (like a data frame), JSON allows you to organize elements into key–value pairs similar to an R list! This allows the data to have much more complex structure, which is useful for web services, but can be challenging for data programming.

In JSON, lists of key–value pairs (called objects) are put inside braces ({}), with the key and the value separated by a colon (:) and each pair separated by a comma (,). Key–value pairs are often written on separate lines for readability, but this isn’t required. Note that keys need to be character strings (so, “in quotes”), while values can either be character strings, numbers, booleans (written in lowercase as true and false), or even other lists! For example:

```
{
    "first_name": "Ada",
    "job": "Programmer",
    "salary": 78000,
    "in_union": true,
    "favorites": {
        "music": "jazz",
        "food": "pizza",
    }
}
```

The above JSON object is equivalent to the following R list:

```r
# Represent the sample JSON data (info about a person) as a list in R
list(
    first_name = "Ada",
    job = "Programmer",
    salary = 78000,
    in_union = TRUE,
    favorites = list(music = "jazz", food = "pizza") # nested list in the list!
)
```

Additionally, JSON supports arrays of data. Arrays are like untagged lists (or vectors with different types), and are written in square brackets ([ ]), with values separated by commas. For example:

```
["Aardvark", "Baboon", "Camel"]
```

which is equivalent to the R list:

```
list("Aardvark", "Baboon", "Camel")
```
Just as R allows you to have nested lists of lists, JSON can have any form of nested objects and arrays. This structure allows you to store arrays (think vectors) within objects (think lists), such as the following (more complex) set of data about Ada:

```json
{
    "first_name": "Ada",
    "job": "Programmer",
    "pets": ["Magnet", "Mocha", "Anni", "Fifi"],
    "favorites": {
        "music": "jazz",
        "food": "pizza",
        "colors": ["green", "blue"]
    }
}
```

The JSON equivalent of a data frame is to store data as an array of objects. This is like having a list of lists. For example, the following is an array of objects of FIFA Men's World Cup data:

```json
[{
    "country": "Brazil", "titles": 5, "total_wins": 70, "total_losses": 17},
{"country": "Italy", "titles": 4, "total_wins": 66, "total_losses": 20},
{"country": "Germany", "titles": 4, "total_wins": 45, "total_losses": 17},
{"country": "Argentina", "titles": 2, "total_wins": 42, "total_losses": 21},
{"country": "Uruguay", "titles": 2, "total_wins": 20, "total_losses": 19}
]
```

You could think of this information as a list of lists in R:

```r
# Represent the sample JSON data (World Cup data) as a list of lists in R
list(
    list(carry = "Brazil", titles = 5, total_wins = 70, total_losses = 17),
    list(carry = "Italy", titles = 4, total_wins = 66, total_losses = 20),
    list(carry = "Germany", titles = 4, total_wins = 45, total_losses = 17),
    list(carry = "Argentina", titles = 2, total_wins = 42, total_losses = 21),
    list(carry = "Uruguay", titles = 2, total_wins = 20, total_losses = 19)
)
```

This structure is incredibly common in web API data: as long as each object in the array has the same set of keys, then you can easily consider this structure to be a data frame where each object (list) represents an observation (row), and each key represents a feature (column) of that observation. A data frame representation of this data is shown in Figure 14.6.

**Remember:** In JSON, tables are represented as lists of rows, instead of a data frame's list of columns.

---

10 FIFA World Cup data: https://www.fifa.com/fifa-tournaments/statistics-and-records/worldcup/teams/index.html
14.4 Processing JSON Data

When working with a web API, the usual goal is to take the JSON data contained in the response and convert it into an R data structure you can use, such as a list or data frame. This will allow you to interact with the data by using the data manipulation skills introduced in earlier chapters. While the *httr* package is able to parse the JSON body of a response into a list, it doesn’t do a very clean job of it (particularly for complex data structures).

A more effective solution for transforming JSON data is to use the *jsonlite* package. This package provides helpful methods to convert JSON data into R data, and is particularly well suited for converting content into data frames.

As always, you will need to install and load this package:

```r
install.packages("jsonlite") # once per machine
library("jsonlite") # in each relevant script
```

The *jsonlite* package provides a function called `fromJSON()` that allows you to convert from a JSON string into a list—or even a data frame if the intended columns have the same lengths!

---

11Package *jsonlite*: full documentation for *jsonlite*: https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf
# Make a request to a given `uri` with a set of `query_params`
# Then extract and parse the results

# Make the request
response <- GET(uri, query = query_params)

# Extract the content of the response
response_text <- content(response, "text")

# Convert the JSON string to a list
response_data <- fromJSON(response_text)

Both the raw JSON data (response_text) and the parsed data structure (response_data) are shown in Figure 14.7. As you can see, the raw string (response_text) is indecipherable. However, once it is transformed using the fromJSON() function, it has a much more operable structure.

The response_data will contain a list built out of the JSON. Depending on the complexity of the JSON, this may already be a data frame you can View()—but more likely you will need to explore the list to locate the “main” data you are interested in. Good strategies for this include the following techniques:

- Use functions such as is.data.frame() to determine whether the data is already structured as a data frame.
- You can print() the data, but that is often hard to read (it requires a lot of scrolling).
- The str() function will return a list’s structure, though it can still be hard to read.
- The names() function will return the keys of the list, which is helpful for delving into the data.

Figure 14.7 Parsing the text of an API response using fromJSON(). The untransformed text is shown on the left (response_text), which is transformed into a list (on the right) using the fromJSON() function.
14.4 Processing JSON Data

As an example continuing the previous code:

```r
# Use various methods to explore and extract information from API results

# Check: is it a data frame already?
is.data.frame(response_data) # FALSE

# Inspect the data!
str(response_data) # view as a formatted string
names(response_data) # "href" "items" "limit" "next" "offset" "previous" "total"

# Looking at the JSON data itself (e.g., in the browser),
# "items" is the key that contains the value you want

# Extract the (useful) data
items <- response_data$items # extract from the list
is.data.frame(items) # TRUE; you can work with that!
```

The set of responses—GitHub repositories that match the search term “dplyr”—returned from the request and stored in the `response_data$items` key is shown in Figure 14.8.

### 14.4.2 Flattening Data

Because JSON supports—and in fact encourages—nested lists (lists within lists), parsing a JSON string is likely to produce a data frame whose columns are themselves data frames. As an example of what a nested data frame may look like, consider the following code:

```r
# A demonstration of the structure of "nested" data frames

# Create a `people` data frame with a `names` column
people <- data.frame(names = c("Ed", "Jessica", "Keagan"))
```

![Figure 14.8](image.png) Data returned by the GitHub API: repositories that match the term “dplyr” (stored in the variable `response_data$items` in the code example).
# Create a data frame of favorites with two columns
favorites <- data.frame(
    food = c("Pizza", "Pasta", "Salad"),
    music = c("Bluegrass", "Indie", "Electronic")
)

# Store the second data frame as a column of the first -- A BAD IDEA
people$favorites <- favorites # the `favorites` column is a data frame!

# This prints nicely, but is misleading
print(people)
# names favorites.food favorites.music
# 1  Ed  Pizza   Bluegrass
# 2  Jessica  Pasta   Indie
# 3  Keagan  Salad   Electronic

# Despite what RStudio prints, there is not actually a column `favorites.food`
people$favorites.food # NULL

# Access the `food` column of the data frame stored in `people$favorites`
people$favorites$food # [1] Pizza Pasta Salad

Nested data frames make it hard to work with the data using previously established techniques and syntax. Luckily, the jsonlite package provides a helpful function for addressing this issue, called flatten(). This function takes the columns of each nested data frame and converts them into appropriately named columns in the “outer” data frame, as shown in Figure 14.9:

# Use `flatten()` to format nested data frames
people <- flatten(people)
people$favorites.food # this just got created! Woo!

Note that flatten() works on only values that are already data frames. Thus you may need to find the appropriate element inside of the list—that is, the element that is the data frame you want to flatten.

In practice, you will almost always want to flatten the data returned from a web API. Thus, your algorithm for requesting and parsing data from an API is this:

1. Use GET() to request the data from an API, specifying the URI (and any query parameters).
2. Use content() to extract the data from your response as a JSON string (as “text”).
3. Use fromJSON() to convert the data from a JSON string into a list.
4. Explore the returned information to find your data of interest.
5. Use flatten() to flatten your data into a properly structured data frame.
6. Programmatically analyze your data frame in R (e.g., with dplyr).
14.5 APIs in Action: Finding Cuban Food in Seattle

This section uses the Yelp Fusion API\(^{12}\) to answer the question:

"Where is the best Cuban food in Seattle?"

Given the geographic nature of this question, this section builds a map of the best-rated Cuban restaurants in Seattle, as shown in Figure 14.12. The complete code for this analysis is also available online in the book’s code repository.\(^{13}\)

To send requests to the Yelp Fusion API, you will need to acquire an API key. You can do this by signing up for an account on the API’s website, and registering an application (it is common for APIs to require you to register for access). As described earlier, you should store your API key in a separate file so that it can be kept secret:

```r
# Store your API key in a variable: to be done in a separate file # (i.e., "api_key.R")
yelp_key <- "abcdef123456"
```

This API requires you to use an alternative syntax for specifying your API key in the HTTP request—instead of passing your key as a query parameter, you’ll need to add a header to the request that you make to the API. An HTTP header provides additional information to the server about who is sending the request—it’s like extra information on the request’s envelope. Specifically,

---

\(^{12}\) Yelp Fusion API documentation: https://www.yelp.com/developers/documentation/v3

\(^{13}\) APIs in Action: https://github.com/programming-for-data-science/in-action/tree/master/apis
you will need to include an “Authorization” header containing your API key (in the format expected by the API) for the request to be accepted:

```
# Load your API key from a separate file so that you can access the API:
source("api_key.R") # the `yelp_key` variable is now available

# Make a GET request, including your API key as a header
response <- GET(
  uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)
```

This code invokes the add_headers() method inside the GET() request. The header that it adds sets the value of the Authorization header to “bearer yelp_key”. This syntax indicates that the API should grant authorization to the bearer of the API key (you). This authentication process is used instead of setting the API key as a query parameter (a method of authentication that is not supported by the Yelp Fusion API).

As with any other API, you can determine the URI to send the request to by reading through the documentation. Given the prompt of searching for Cuban restaurants in Seattle, you should focus on the Business Search documentation, a section of which is shown in Figure 14.10.

![Business Search Documentation](https://www.yelp.com/developers/documentation/v3/business_search)

Figure 14.10  A subset of the Yelp Fusion API Business Search documentation.

---

As you read through the documentation, it is important to identify the query parameters that you need to specify in your request. In doing so, you are mapping from your question of interest to the specific R code you will need to write. For this question (“Where is the best Cuban food in Seattle?”), you need to figure out how to make the following specifications:

- **Food**: Rather than search all businesses, you need to search for only restaurants. The API makes this available through the `term` parameter.
- **Cuban**: The restaurants you are interested in must be of a certain type. To support this, you can specify the category of your search (making sure to specify a supported category, as described elsewhere in the documentation\(^{15}\)).
- **Seattle**: The restaurant you are looking for must be in Seattle. There are a few ways of specifying a location, the most general of which is to use the `location` parameter. You can further limit your results using the `radius` parameter.
- **Best**: To find the best food, you can control how the results are sorted with the `sort_by` parameter. You’ll want to sort the results before you receive them (that is, by using an API parameter and not `dplyr`) to save you some effort and to make sure the API sends only the data you care about.

Often the most time-consuming part of using an API is figuring out how to hone in on your data of interest using the parameters of the API. Once you understand how to control which resource (data) is returned, you can then construct and send an HTTP request to the API:

```r
# Construct a search query for the Yelp Fusion API's Business Search endpoint
base_uri <- "https://api.yelp.com/v3"
endpoint <- "/businesses/search"
search_uri <- paste0(base_uri, endpoint)

# Store a list of query parameters for Cuban restaurants around Seattle
query_params <- list(
  term = "restaurant",
  categories = "cuban",
  location = "Seattle, WA",
  sort_by = "rating",
  radius = 8000 # measured in meters, as detailed in the documentation
)

# Make a GET request, including the API key (as a header) and the list of
# query parameters
response <- GET(
  search_uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)
```

\(^{15}\) **Yelp Fusion API Category List**: https://www.yelp.com/developers/documentation/v3/all_category_list
As with any other API response, you will need to use the `content()` method to extract the content from the response, and then format the result using the `fromJSON()` method. You will then need to find the data frame of interest in your response. A great way to start is to use the `names()` function on your result to see what data is available (in this case, you should notice that the `businesses` key stores the desired information). You can `flatten()` this item into a data frame for easy access.

```r
# Parse results and isolate data of interest
response_text <- content(response, type = "text")
response_data <- fromJSON(response_text)

# Inspect the response data
names(response_data) # [1] "businesses" "total" "region"

# Flatten the data frame stored in the `businesses` key of the response
restaurants <- flatten(response_data$businesses)
```

The data frame returned by the API is shown in Figure 14.11.

Because the data was requested in sorted format, you can `mutate` the data frame to include a column with the rank number, as well as add a column with a string representation of the name and rank:

```r
# Modify the data frame for analysis and presentation
# Generate a rank of each restaurant based on row number
restaurants <- restaurants %>%
  mutate(rank = row_number()) %>%
  mutate(name_and_rank = paste0(rank, ". ", name))
```

The final step is to create a map of the results. The following code uses two different visualization packages (namely, `ggmap` and `ggplot2`), both of which are explained in more detail in Chapter 16.

![Figure 14.11](image)

A subset of the data returned by a request to the Yelp Fusion API for Cuban food in Seattle.
14.5 APIs in Action: Finding Cuban Food in Seattle

Figure 14.12 A map of the best Cuban restaurants in Seattle, according to the Yelp Fusion API.

```r
# Create a base layer for the map (Google Maps image of Seattle)
base_map <- ggmap(get_map(location = "Seattle, WA", zoom = 11))

# Add labels to the map based on the coordinates in the data
base_map +
  geom_label_repel(
    data = response_data,
    aes(x = coordinates.longitude, y = coordinates.latitude, label = name_and_rank)
  )
```

Below is the full script that runs the analysis and creates the map—only 52 lines of clearly commented code to figure out where to go to dinner!
Chapter 14  Accessing Web APIs

# Yelp API: Where is the best Cuban food in Seattle?
library("httr")
library("jsonlite")
library("dplyr")
library("ggrepel")
library("ggmap")

# Load API key (stored in another file)
source("api_key.R")

# Construct your search query
base_uri <- "https://api.yelp.com/v3/"
endpoint <- "businesses/search"
uri <- paste0(base_uri, endpoint)

# Store a list of query parameters
query_params <- list(
  term = "restaurant",
  categories = "cuban",
  location = "Seattle, WA",
  sort_by = "rating",
  radius = 8000
)

# Make a GET request, including your API key as a header
response <- GET(
  uri,
  query = query_params,
  add_headers(Authorization = paste("bearer", yelp_key))
)

# Parse results and isolate data of interest
response_text <- content(response, type = "text")
response_data <- fromJSON(response_text)

# Save the data frame of interest
eats <- flatten(response_data$businesses)

# Modify the data frame for analysis and presentation
eats <- eats %>%
  mutate(rank = row_number()) %>%
  mutate(name_and_rank = paste0(rank, ". ", name))

# Create a base layer for the map (Google Maps image of Seattle)
base_map <- ggmap(get_map(location = "Seattle, WA", zoom = 11))
# Add labels to the map based on the coordinates in the data
base_map +
  geom_label_repel(
    data = restaurants,
    aes(x = coordinates.longitude, y = coordinates.latitude, label = name_and_rank)
  )

Using this approach, you can use R to load and format data from web APIs, enabling you to analyze and work with a wider variety of data. For practice working with APIs, see the set of accompanying book exercises.¹⁶

¹⁶API exercises: https://github.com/programming-for-data-science/chapter-14-exercises
This page intentionally left blank
Index

Symbols

, (comma)
  data frame syntax, 122
  function syntax, 69
  key-value pair syntax, 191

" (double quotes), character data syntax, 61
’ (single quotes), character data syntax, 61
.. (double dot), moving up directory, 14
. (single dot), referencing current folder, 14

| (pipe)
  directing output, 20
  pipe table, 48

! (exclamation point), Markdown image syntax, 47

# (pound/hashtag symbol)
  comment syntax, 10, 58

$ (dollar notation)
  accessing data frames, 122
  accessing list elements, 97–98

%>% (pipe operator), dplyr package, 141–142

() (parentheses)
  function syntax, 70
  Markdown hyperlink syntax, 46

* (asterisk wildcard)
  loading entire table from database, 173
  using wildcards with files, 17–18

? (question mark), query parameter syntax, 184

[] (single-bracket notation)
  accessing data frames, 122–123
  comparing single- and double-bracket notation, 101
  Markdown hyperlink syntax, 46
  retrieving value from vector, 88

[[ ]] (double-bracket notation)
  list syntax, 98–99, 101
  selecting data of interest for application, 312
\{\} (braces)

\{\} (braces)
  code chunk syntax, 279
  key-value pair syntax, 191
  render function syntax, 308

<- (assignment operator), 59, 92
>> directing output, 20
> directing output, 20
- (tilde), home directory shorthand, 10, 15

Absolute path
  for CSV data, 125
  finding R and Rscript, 57
  for images, 48
  specifying paths, 14-15
  URLs and, 47
Access tokens (API keys)
  example finding Cuban food in Seattle, 196–197
  registering with web services, 186–188
add (git). See also Staging Area
  add and commit changes, 38–39, 322, 327–328, 333, 337
  adding files to repository, 32–33
  unadd, 35
aes() function, for aesthetic mappings, 237
Aesthetics
  adding titles and labels to charts, 246
  aesthetic mappings, 234, 237–238
  data visualization, 229–230
Aggregation
  proportional representation of data and, 212–213
  in Shiny example, 315–316
  statistical transformation of data, 255
  using summarize(), 138–139
Analysis. See Data analysis
Annotation
  capabilities of version control systems, 28
  ggplot2 package, 246–248
Anonymous variables, 71, 140
anscombe data set, in R, 208
Anscombe’s Quartet, 208
API keys (Access tokens)
  example finding Cuban food in Seattle, 196–197
  registering with web services, 186–188
APIs (application programming interfaces).
  See also Web APIs
  defined, 181
  in plotly package, 258

Application servers, developing, 306–309
Applications
  Shiny app example applying to fatal police shootings, 311–318
  structure in Shiny framework, 295–299
app.R file, 295–296
Apps, publishing Shiny, 309–311
Area encoding, visualizing hierarchical data, 218
Arguments
  commands and, 13
  creating data frames, 120
  creating lists, 96
  debugging functions, 78
  function inputs, 69–70
  function parts, 76
  named arguments, 72–73
  syntax of, 16
  vectorized functions and, 87
arrange()
  dplyr core functions, 131, 137–138
  summarizing information using dplyr functions, 313
Arrays, JSON support, 191–192
AS keyword, renaming columns, 173
Assignment operator (<-)
  assigning values to variables, 59
  modifying vectors, 92
Atom
  preview rendering support, 49–50
  selecting text editor, 6–7
  writing code, 3
Authentication, API authentication service, 187

B
Bar charts
  facets and, 245
  position adjustments, 240
  proportional representation of data, 211–213
  visualizing data with single variable, 210–211
Bash shell. See also Git Bash
  commands, 13
  executing code, 4
  ls command, 13
Bins, breaking data into different variables, 142
BitBucket, comparing with GitHub, 29
Blockquotes, markdown options, 48
Blocks, markdown formatting syntax, 47
Body, function parts, 76–77
Bracket notation
  double. See \[
  \] (double-bracket notation)
  retrieving value from vector using bracket notation, 88
  single. See \[\] (single-bracket notation)

Branches
  \texttt{git} branching model, 319–320
  merging, 324–325
  merging from GitHub, 328–329
  resolving merge conflicts, 327–328
  tracking code versions with, 319–320
  using in feature branch workflows, 333–335
  using in forking workflows, 335–339
  working with, 320–324
  working with feature branches, 329–331

Checkpoints. See Commit

Choropleth maps
  drawing and examples, 248–251
  overview of, 248

Chunks
  breaking data into different variables, 142
  inline code and, 280
  options, 279–280
  .Rmd files and, 277–278

Circle packing, visualizing hierarchical data, 218–219

clone (\texttt{git})
  collaboration using forking workflow, 336
  creating centralized repository, 332
  forks, 337
  merging branches and, 328
  repos, 36–39, 43
  understanding/using \texttt{git} commands, 43

Code
  chunks, 142, 277–280
  executing, 4–5
  inline code, 280
  managing, 3–4
  running, 54–57
  syntax-colored code blocks, 48
  tracking versions with branches, 319–320
  Visual Studio Code (VS Code), 7, 49
  writing, 3

Collaboration
  centralized workflow for, 331
  creating centralized repository, 331–333
  interactive web applications and. See Shiny framework
  merging branches, 324–325, 328–329
  overview of, 273–274, 319
  reports. See R Markdown
  resolving merge conflicts, 327–328
  tracking code versions, 319–320
  working with branches, 320–324
  working with feature branches, 329–331, 333–335
  working with forking workflows, 335–339

\texttt{collect()}, manipulating table data, 177–178

Colon operator (a:b)
  creating vectors, 82
  specifying range of vector index, 90

Color
  adding to Leaflet map, 270
  color palettes, 223–225, 242
  effective for data visualization, 222–226
  \texttt{ggplot2} color scales, 242–243
ColorBrewer tool
   color palettes, 242
   examples, 289
   overview of, 223–225

colorFactor(). Leaflet maps, 270

Columns
   changing to/from rows using tidyR, 157–159
   dplyr arrange() operation, 137–138
   dplyr filter() operation, 135
   dplyr mutate() operation, 136

Columns (fields), in relational databases, 168

Comma-separated value data. See CSV (comma-separated value) data

Command line
   accessing, 9–10
   changing directories, 12–13
   cloning repository, 37
   commit history, 320
   directing/redirecting output, 20
   executing code, 4
   handling errors, 18–19
   interacting with databases, 31
   learning new commands, 16–17
   listing files, 13
   managing files, 15–16
   navigating files, 11–12
   networking commands, 20–23
   overview of, 9
   running R code, 56–57
   set up tools, 4–5
   specifying paths, 14–15
   wildcards, 17–18
   working with, 4

Command Prompt. See Command line

Command Prompt (Windows)
   accessing, 9–10
   executing code, 4
   working with, 5

Command shell (terminal). See Command line

Commands. See also by individual types
   issuing, 13
   list of advanced, 18
   list of basic, 15

Comments
   R language, 58
   syntax for code comments, 10

commit (git)
   add and commit changes, 33, 38–39, 327–328, 337
   creating centralized repository, 333
   git core concepts, 28
   history, 40
   message etiquette, 34–35
   reverting to earlier versions, 40–42
   tracking code versions, 319–320
   understanding/using git commands, 43
   working with branches, 320–324
   working with feature branches, 330–331, 334

Communities
   resources for learning R, 66–67
   sources of data, 109

Comparison operators, logical values and, 62

Compiled languages, 53

Complex data type, 63, 99

Comprehensive R Archive (CRAN), 6

Computer, set up, 3–4

Concurrency, capabilities of version control systems, 28

Conditional statements, 79–80

config, configuring git for first-time use, 30

Console, RStudio, 55

Content
   building Shiny application, 313
   content elements in designing UIs, 299
   extracting from HTTP request, 200
   static content in Shiny framework, 300–301

content(), extracting content from HTTP request, 200

Continuous color scales, 225–226

Continuous data
   choosing effective colors for data visualization, 223
   selecting visual layouts, 209–210
   visualization with multiple variables, 213–216
   visualizing with single variable, 210

Control widgets
   developing application servers, 307
   in Shiny framework, 298
   user interactions in Shiny apps, 301–303

coord() functions
   coord_flip() example, 244
   types of coordinate systems for geometric objects, 243–244

Coordinate systems
   coord_flip() example, 244
   creating choropleth maps, 249–250
   creating dot distribution maps, 252
   Grammar of Graphics, 232
   types for geometric objects, 243–244

cor(), correlation function in R, 161

count(), summarizing information, 313

courses(), resources for learning R, 65–66

CRAN (Comprehensive R Archive), 6

CSS language, 342

CSV (comma-separated value) data
   factor variables, 126–129
Data

acquiring domain knowledge, 112–113
analyzing, See Data analysis
answering questions, 116–118
dplyr example analyzing flight data, 148–153
dplyr grammar for manipulating, 131-132
encoding, 220–222, 229, 237
finding, 108-109
flattening JSON data, 196–197
generating, 107-108
interactive presentation, 293
interpreting, 112
measuring, 110-111
overview of, 107
ratio data, 111
reusable functions in managing, 70
schemas, 113-116
structures, 111-112, 122
transforming into information, 341
understanding data schemas, 113-116
visualization of, See Data visualization
wrangling, 106

Data analysis

generating data, 108
reusable functions, 70
tidy package. See tidy package

Data frames

accessing, 122–123
analyzing by group, 142-144
creating, 120-121
describing structure of, 121-122
factor variables, 126-129
joining, 144–148
overview of, 119-120
viewing working directory, 125–126
working with CSV data, 124–125
data() function, viewing available data sets, 124–125

Data-ink ratio, aesthetics of graphics, 229

Data structures

overview of, 111–112
two-dimensional, 122

Data types

factors, 120
lists and, 95
R language, 60–63
selecting visual layouts, 209–210
vectorized functions and, 87
vectorized operations and, 83

Data visualization

aesthetics, 229–230
choosing effective colors, 222–226
choosing effective graphical encodings, 220–222
expressive displays, 227–229
ggplot2. See gggplot2 package
of hierarchical data, 217–220
leveraging preattentive attributes, 226–227
with multiple variables, 213–217
overview of, 205–207
purpose of, 207–209
reusable functions, 70
selecting visual layouts, 209–210
with single variable, 210–213
tidy package. See tidy package

Data visualization, interactive

example exploring changes to Seattle, 266–272
leaflet package, 263–266
overview of, 257–258
plotly package, 258–261
rbokeh package, 261–263

Databases

accessing from R, 175–179
designing relational, 144
overview of relational, 167–169
setting up relational, 169–171
SQL statements, 171–175

DataCamp, resources for learning R, 66
dbConnect(), accessing SQLite, 176–177
dbListTables(), listing database tables, 177
dplyr package, 176–179
dplyr package, accessing databases, 174

Debugging functions, 78. See also Error handling

Directories

accessing command line and, 10
changing from command line, 12-13
printing working directory, 11
Directories

- tree structure of, 12
- turning into a repository, 31
- viewing working directory, 125–126

Displays, expressive, 227–229

Distributions, of x and y values (statistics), 208–209

Documentation
- of commands, 16
- getting help via, 64
- resources for learning R, 66
- Shiny layouts, 304

Documents
- creating, 275
- knitting, 278

Domain, interpreting data by, 112–113

Dot distribution maps, 248, 251–252

Double-bracket notation. See [][ ] (double-bracket notation)

**dplyr package**

- analyzing data frames, 142–144
- analyzing flight data, 148–153
- arrange(), 137–138
- converting dplyr functions into SQL statements, 178
- core functions, 131–132
- example mapping evictions in San Francisco, 252
- example report on life expectancy, 289
- filter(), 135–136
- grammar for data manipulation, 131–132
- group_by(), 244
- joining data frames, 144–148
- mutate(), 136–137
- orienting data frames for plotting, 239
- overview of, 131
- performing sequential operations, 139–141
- pipe operator ( %>% ), 141–142
- select(), 133–134
- summarize(), 138–139

Dynamic inputs, Shiny framework, 301–303

Dynamic outputs, Shiny framework, 303–304

Dynamically typed languages, 60

Error handling
- command line, 18–19
- debugging functions, 78
- reading error messages, 63

Ethical responsibilities, 343

Excel, working with CSV data, 124

**exit**

- disconnecting from remote computer, 22
- stopping or canceling program or running command, 19

Expressions, multiple operators in, 61

Extensions, file, 6, 48–49

**F**

facets, 244–245

Facets

- ggplot2 package, 244–245
- Grammar of Graphics, 232

Factors

- creating data frames, 120
- variables, 126–129

Feature branches

- in centralized workflow, 333–335
- working with, 329–331

Fields (columns), in relational databases, 168

**figure()**, creating Bokeh plots, 262–263

Files

- adding to repository, 32–33
- changing directories, 12–13
- creating .Rmd files, 276–278
- extensions, 6, 48–49
- ignoring, 42–44
- listing, 13
- managing, 15–16
- navigating, 11–12
- specifying paths, 14–15

**fill()**, aesthetic layouts, 238–240

filter()

- dplyr core functions, 131, 135–136
- example report on life expectancy, 289
- manipulating table data, 177–178

Filtering

- joins, 148
- vectors, 90–91, 93

**flatten()**

- example finding Cuban food in Seattle, 200, 202
- JSON data, 196–197

for loops, 87

Foreign keys, in relational databases, 168–169

fork, repos on GitHub, 36–38
Forking workflow
  feature branches in, 331, 333–335
  working with, 335–339

Formats
  table, 157
  text, 46

Formulas, 245

Frameworks
  defined, 293
  Shiny framework. See Shiny framework

fromJson(), converting JSON string to list, 193–194, 200

full_join(), 148

function keyword, 76

Functions
  for aesthetic mappings (aes()), 237–238
  applying to lists, 102–103
  built-in, 71–72
  c() function, 81–82
  conditional statements, 79–80
  converting dplyr functions into SQL statements, 178
  coord_ functions, 243–244
  correlation function (cor()), 161
  creating lists, 96
  debugging, 78. See also Error handling
devloping application servers, 307–309
  geometry. See geom_ functions
  inspecting data frames, 121–122
  loading, 73–75
  named arguments, 72–73
  nested statements within, 140–141
  overview of, 69–70
  referencing database table, 177
  in Shiny layouts, 305
  syntax, 70–71
tidyr functions for changing columns to/from rows, 157–159
  vectorized, 86–88
  viewing available data sets (data()), 124–125
  writing, 75–77

Functions, dplyr
  arrange(), 137–138
  core functions, 131–132
  filter(), 135–136
  group_by(), 142–144
  left_join(), 145–147
  mutate(), 136–137
  overview of, 132
  select(), 133–134
  summarize(), 138–139
  summarizing information using, 313

getwd(), viewing working directory, 125

ggmap package
  example finding Cuban food in Seattle, 200–203
  example mapping evictions in San Francisco, 253
  map tiles, 252

ggplot()
  creating plots, 232, 234
  example mapping evictions in San Francisco, 256

ggplot2 package
  aesthetic mappings, 237–238
  basic plotting, 232–235
  choropleth maps, 248–251
  coordinate systems, 243–244
  dot distribution maps, 252
  example finding Cuban food in Seattle, 200
  example mapping evictions in San Francisco, 252–256
  facets, 244–245
  Grammar of Graphics, 231–232
  labels and annotations, 246–248
  map types, 248
  position adjustments, 238–240
  rendering plots, 284
  specifying geometries, 235–237

GET
  example finding Cuban food in Seattle, 197–198, 202
  HTTP verbs, 188–189
  sending GET requests, 189–190

geom_ functions
  adding titles and labels to charts, 247–248
  aesthetic mappings and, 237–238
  creating choropleth maps, 249–250
  creating dot distribution maps, 252
  example mapping evictions in San Francisco, 253–256
  rendering plots, 284
  specifying geometric objects, 234
  specifying geometries, 235–237
  statistical transformation of data, 237

ggplot2 layers, 232
  position adjustments, 238–240
  specifying geometric objects, 234–235
  specifying with
static plot of iris data set, 257–258
statistical transformation of data, 255
styling with scales, 240–242
tidy example, 160–161
ggplotly(), 259
ggrepel package, preventing labels from overlapping, 247–248
git
accessing project history, 40–42
adding files, 32–33
branching model. See Branches
checking repository status, 31–33
committing changes, 33–35
core concepts, 27–28
creating repository, 30–31
ignoring files, 42–44
installing, 5
leveraging using GitHub, 6
local git process, 35
managing code with, 3–4
overview of, 27–28
project setup and configuration, 30
tracking changes, 32
tutorials, 43–44
version control, 4
Git Bash. See also Bash shell
accessing command line, 9–10
commands used by, 13
executing code using Bash shell, 4–5
ls command, 13
tab-completion support, 15
Git Flow model, 335
GitHub
accessing project history, 40–42
creating centralized repository, 331–333
creating GitHub account, 6
forking/cloning repos on GitHub, 36–38
ignoring files, 42–44
managing code with, 3
overview of, 29
pushing/pulling repos on GitHub, 38–40
README file, 48–49
sharing reports as website, 285–286
storing projects on, 36
tutorials, 43–44
.gitignore, ignoring files, 42–44
GitLab, comparing with GitHub, 29

Google Docs, version control systems compared with, 28
Google, getting help via, 63
Google Sheets, working with CSV data, 124
Government publications, sources of data, 108
Grammar of Data Manipulation (Wickham), 131
Grammar of Graphics, 231–232

Graphics. See also by individual types of graphs: Data visualization

aesthetics, 229–230
choosing effective graphical encodings, 220–222
expressive displays, 227–229
with ggplot2. See ggplot2 package
Grammar of Graphics, 231–232
leveraging preattentive attributes, 226–227
selecting visual layouts, 209–210
visualizing hierarchical data, 217–220
group_by(),
analyzing data frames by group, 142–144
facets and, 244
statistical transformation of data, 255
summarizing information using, 313
GROUP_BY clause, SQL SELECT, 174

Heatmaps. See also Choropleth maps
data visualization with multiple variables, 215, 217
example mapping evictions in San Francisco, 256
Help
R language, 63–64
RStudio, 55
Hidden files, 42–44
Hierarchical data, visualization of, 217–220

Histograms
data visualization with multiple variables, 216
expressive displays, 229
visualizing data with single variable, 210

Hosts, Shiny apps, 309–310

HSL Calculator, 223
HSL (hue-saturation-lightness) color model, 222–223

HTML (Hypertext Markup Language)

HTML Tags Glossary, 300–301
markup languages, 45
sharing reports as website, 284–286
web development language, 342

HTTP (HyperText Transfer Protocol)
header, 196–197
overview of, 181–182
verbs, 188–189

HTTP requests
example finding Cuban food in Seattle, 196–200
response header and body, 190
web services and, 181

HTTP verbs, Web APIs, 188–189

http package
parsing JSON data, 192–193
sending GET requests, 189–190

Hue
choosing effective colors for data visualization, 222
multi-hue color scales, 225
Hue-saturation-lightness (HSL) color model, 222–223

Hyperlinks, markdown, 46–47

Icons, types of interfaces, 9
IDE (integrated development environment), 54
if_else, conditional statements, 79–80
Images, markdown, 47–48
Indices
for getting subsets of vectors, 88–89
multiple indices, 89–90
init (git), turning a directory into a git repository, 31

Inline code, in R Markdown, 280

INNER JOIN clause, SQL SELECT, 174
inner_join(), 147–148

Inputs
dynamic inputs with Shiny framework, 301–303
functions and, 69
Shiny framework, 293–294

Integer data type, 63

Integrated development environment (IDE), 54

Interactivity
interactive data visualization. See Data visualization, interactive
interactive web applications. See Shiny framework

Interface
command line as, 9
defined, 181
user. See UIs (user interfaces)
web APIs. See Web APIs

Interpreted languages, 53
Interval data, measuring data, 111
iris data set, interactive plots in, 257–258

Italics, text formatting, 45–46

JavaScript, 342–343
join()
dplyr core functions, 131
joining data frames, 144–148
JOIN clause, SQL SELECT, 174–175

Journalism, sources of data, 109
JSON (JavaScript Object Notation)
flattening JSON data, 195–197
list of lists structure in, 97
parsing JSON data, 193–195
processing JSON data, 191–193
jsonlite package, 192–193

kable(), knitr package, 283–284, 291

Key-value pairs
JSON (JavaScript Object Notation), 191
query parameters and, 184
tidyR data tables, 157

knitr package
creating R Markdown documents, 275
kable(), 283–284, 291

Knitting documents, 278

Labels
adding to plots, 246–248
aesthetics of graphics, 230
labs(), adding titles and labels to charts, 246
lapply(), applying functions to lists, 102–103

Layers, ggplot2 package, 232
layout(), 260–261, 268

Layouts
coordinate systems, 243–244
designing UIs, 299
example exploring changes to Seattle, 268
facets, 244–245
labels and annotations, 246–248
plotly package, 260–261
position adjustments, 238–240
selecting visual, 209–210
Shiny framework, 304–306
styling with scales, 240–242

Lazy evaluation, in dplyr package, 178

leaflet()
creating Leaflet map, 264
dynamic inputs with Shiny framework, 268
example exploring changes to Seattle, 269
leaflet package

creating interactive plots, 264–266
example exploring changes to Seattle, 269–271
installing and loading, 263
Shiny app example applying to fatal police shootings, 312–313
Learn Git Branching, 339
LEFT JOIN clause, SQL SELECT, 174
left_join()
example of join operation, 145–146
join types, 146–147
Legends
adding to Leaflet map, 270–271
aesthetics of graphics, 230
length() function, determining number of elements in a vector, 82
Libraries. See Packages
library(), referencing external packages, 311
Lightness, choosing effective colors for data visualization, 223
Linux
command-line tools on, 5
installing git, 5
list() function, creating lists, 96
Lists
accessing elements of, 97–99
applying functions to, 102–103
converting JSON string to list, 193–194
creating, 96–97
creating data frames, 120–121
double-bracket notation, 101
JSON structures compared with, 192–193
listing files from command line, 13
modifying, 100
overview of, 95
rendering Markdown lists, 282–283
log, viewing commit history, 40
Logical (boolean)
data type, 61–63
debugging functions, 78
operators, 62–63
vector filtering by values, 90–91
Loops, vectorized functions and, 87
ls
list folder contents, 13
using with remote computer, 22
-M option, adding messages to commit command, 34
Mac OSs. See also Terminal (Mac)
accessing command line, 9–10
command-line tools on, 4
installing git, 5
Machine learning, making predictions, 342
Mackinlay's Expressiveness Criteria, 227–229
man, looking up commands in manual, 16–17
Map tiles
adding to Leaflet map, 264
ggmap package, 252
Maps
aesthetic mappings, 237–238
choropleth maps, 248–251
dot distribution maps, 251–252
equivalent mapping evictions in San Francisco, 252–256
interactive, 263
types of, 248
Markdown
hyperlinks, 46–47
images, 47–48
overview of, 45
rendering, 48–50
rendering lists, 282–283
rendering strings, 281
rendering tables, 283–284
static content elements of UIs, 300–301
tables, 48
text formatting and blocks, 46
Markdown Reader, 49
Markers, adding to Leaflet map, 264
Markup languages, 45
Mathematical operators
applying to vectors, 83
assigning values to variables, 59
using on numeric data types, 60
vectorized functions and, 86–87
Matrix, two-dimensional data structures in R, 122
.md file extension, for markdown files, 48
Menus, types of interfaces, 9
merge (git)
combining branches, 324–325
forking/cloning repository on GitHub, 337–338
resolving merge conflicts, 327–328
working with feature branches, 330, 334–335
Merging. git core concepts, 29
message etiquette, commit. 34–35
Meta-data, 114–116, 277
Microsoft Excel, 124
Microsoft Windows. See Windows OSs
mkdir, documentation of commands, 16–17
Moral responsibility, 343
\texttt{mutate()}
do\texttt{lry core functions, 131, 136–137
example finding Cuban food in Seattle, 202
example report on life expectancy, 289–290
Mutating joins, 148
MySQL, 171

\texttt{NA}
value
compared with NULL, 100
logical values and, 89
modifying vectors and, 92

Named argument, \texttt{R} functions, 72–73
Named lists, creating data frames, 120
\texttt{names()}
function, creating lists and, 96
Negative index, vector indices, 89
Nested objects, JSON support, 192
Nested statements, within other functions, 140–141
Nested structures, visualizing hierarchical data, 217–220
Networking commands, 20–23

News, sources of data, 109
Nominal (categorical) data
  choosing effective colors for data visualization, 223
  data visualization with multiple variables, 215
  measuring data, 110
  proportional representation of data and, 212
  selecting visual layouts and, 209–210
  visualizing single variable, 210

Non-standard evaluation (NSE), \texttt{dplyr}, 133

\texttt{NULL}
value, modifying lists and, 100

Numbers, working with CSV data, 124
Numeric data type, 60–61, 95

OAuth, API authentication service, 187
Observations, data structures, 111–112
\texttt{ON} clause, SQL SELECT, 174
Online communities, sources of data, 109
Open source, \texttt{R} language as, 53
\texttt{OpenStreetMap}, 264

Operationalization, using data to answer questions, 116–118

Orders of \texttt{R} functions, 73–75
of \texttt{R} functions, 116–118
out-of-bounds indices, vector indices, 89
\texttt{OUTER JOIN} clause, SQL SELECT, 174

Outliers, visualizing data with single variable, 210

Output
  directing/redirection, 20
  dynamic, 303–304
  functions and, 69
  reactive, 295
  Shiny framework, 293–294

Packages
  Bokeh, 261
  dbplyr, 176–179
dplyr. See \texttt{dplyr} package
  \texttt{ggmap}. See \texttt{ggmap} package
  \texttt{ggplot2}. See \texttt{ggplot2} package
  \texttt{ggrepel}, 247–248
  \texttt{httr}, 189–190, 192–193
  \texttt{jsonlite}, 192–193
  \texttt{knitr}. See \texttt{knitr} package
  \texttt{leaflet}. See \texttt{leaflet} package
  \texttt{plotly}, 258–261
  of \texttt{R} functions, 73–75
  \texttt{rbokeh}, 261–263
  \texttt{RColorBrewer}, 224–225
  referencing external, 311
  \texttt{rmarkdown}, 275
  \texttt{RStudio}, 55
  \texttt{tidyr}. See \texttt{tidyr} package
  \texttt{tidyverse}, 132, 142

Panning, interactive data visualization, 257

Parameters
  function inputs, 69–70
  query parameters, 184–186, 202

Passing arguments
  debugging functions, 78
  to functions, 70

\texttt{PATCH}, HTTP verbs, 188

Paths
  finding, 57
  on remote computers, 22
  specifying from command line, 14–15
  viewing working directory, 123

Pie charts, 211–213, 221
pipe operator (\%\%\%), *dplyr* package, 141–142

pipe table, 48

`plot_ly()`

creating plots, 260

example exploring changes to Seattle, 268

`plotly` package

creating interactive plots, 259–261

example exploring changes to Seattle, 268

loading, 258

Plots

`ggplot2` package. See `ggplot2` package

`plotly` package. See `plotly` package

plotting, 232–235

rendering in R Markdown, 284

RStudio, 55

Pointers, types of interfaces, 9

Popups, adding interactivity to Leaflet map, 266

Positional arguments

functions and, 72–73

`ggplot2` geometries, 238–240

PostgreSQL, 170–171, 176

Powershell, Windows Management Framework, 5

Preattentive processing, in data visualization, 226–227

Predictions, 342

Preview Markdown rendering, 49

Primary keys, in relational databases, 168–169

`print()`; analyzing flight data, 152

Probability, 342. See also Statistics

Problem domain, interpreting data by domain, 112–113

Programming/programming languages

compiled languages, 53

data wrangling, 106

dynamically vs. statically typed languages, 60

interpreted languages, 53

learning, 342–343

markup languages, 45

R language. See R language

5 language, 53

SQL. See SQL (Structured Query Language)

statically typed, 60

statistical languages, 53

Proportional representation, visualizing data with single variable, 211–212

publishing apps, Shiny framework, 309–311

`pull (git)`

creating centralized repository, 333

merging from GitHub, 328

repos on GitHub, 38–40

understanding/using `git` commands, 43

working with feature branches, 335

Pull request, GitHub, 335–339

`push (git)`

creating centralized repository, 333

merging from GitHub, 328–329

repos on GitHub, 38–40

understanding/using `git` commands, 43

working with feature branches, 333–335

`pwd`, print working directory, 11, 22

Python, 342

`qmplot()`, creating background maps, 253–254

Query parameters

example finding Cuban food in Seattle, 202

in Web URIs, 184–186

`quit (q)`, stopping or canceling program or running command, 19

R

R for Everyone, 341

R language

accessing databases, 175–179

accessing Web APIs, 189–190

anscombe data set in, 208

arguments, 72–73

built-in functions, 71–72

code chunks and, 279–280

comments, 58

data types, 60–63

downloading, 6–8

as dynamically typed language, 60

function packages, 73–75

function syntax, 70–71

functions in Shiny layouts, 305

help resources, 63–64

interactive data visualization. See Data visualization, interactive

learning, 64–67

overview of, 4

programming with, 53–54

running R code from command line, 56–57

running R code using RStudio, 54–56

two-dimensional data structures, 122

variable definition, 58–60

web application framework. See Shiny framework

R Markdown

code chunks and, 279–280
Ratio data, measuring, 111
rbokeh package
creating interactive plots, 262–263
installing and loading, 261–262
RColorBrewer package, 224–225
RDMS (relational database management system), 169. See also Relational databases
Reactive output
dynamic outputs with Shiny framework, 303–304
render functions and, 308
in Shiny framework, 295
Reactivity, in Shiny framework, 295
read.csv()
creating choropleth maps, 250
example mapping evictions in San Francisco, 253
in R, 161
README file, GitHub, 48–49
Records
data structures, 111–112
keeping, 107–108
Recycling operation, vectors, 84–85
Redirects, output, 20
Relational databases
accessing, 175–179
designing, 144
overview of, 167–169
setting up, 169–171
SQL statements, 171–175
Relational operators
logical values and, 62
vector filtering with, 91
Relationships
assessing in statistical learning, 341–342
between x and y values (statistics), 208–209
Relative path
images, 48
specifying paths, 14
rows (records), in relational databases, 357
URLs, 47
viewing working directory, 123–126
Remote repository
git core concepts, 29
repositories as remotes, 36
Remote computers, accessing, 20–21
Render function
developing application servers, 307–309
in Shiny framework, 295–296
Rendering markdown, 48–50
Reports, 275. See also R Markdown
Repository (repo)
checking status, 31–33
creating, 30–31
creating centralized repository, 331–333
forking/cloning on GitHub, 36–38, 336–337
git core concepts, 28
linking online to local, 36
pushing/pulling on GitHub, 38–40
viewing current branch, 320–321
REpresentational State Transfer. See REST (REpresentational State Transfer)
Required arguments, functions and, 72
Research, sources of data, 109
reset, destroying commit history, 42
Response body, HTTP requests, 190
Response header, HTTP requests, 190
REST (REpresentational State Transfer)
responding to HTTP requests, 189
web APIs, 182
web services and, 181
Return value
c() function, 81–82
function parts, 77
writing functions, 75–76
Reversibility
capabilities of version control systems, 28
reverting to earlier versions, 40–42
revert, reverting to earlier versions, 40–42
RIGHT JOIN clause, SQL SELECT, 174
right_join(), 145–147
rmarkdown package, creating R Markdown documents, 275
.Rmd files, creating, 276–278
round() function, vectorized functions and, 86–87
Rows
arrange() operation, 137–138
changing from columns to/from, 157–159
filter() operation, 135
Rows (records), in relational databases, 168
Sequences, performing sequential operations, 139–141
seq() function, creating vectors and, 82–83

Scientific research, sources of data, 109

Scalable vector graphics (SVGs), 266

Scalar, example adding, 85–86

Scatterplot matrix, 213

Scatterplots
Anscome’s Quartet, 209
data visualization with multiple variables, 213–217
ggplot2 example, 233

Scripts
programming with R language, 53–54
running from command line, 57
running using RStudio, 54

select()
dplyr core functions, 131, 133–134
example report on life expectancy, 289–290
manipulating table data, 177–178

SELECT statement
ON clause, 174
JOIN clause, 174–175
ORDER BY and GROUP BY clauses, 174
SQL statements, 171–174
WHERE clause, 173–174

Sensors, generating data, 107

seq() function, creating vectors and, 82–83

Servers
application structure in Shiny framework, 296
building Shiny application, 313–318
defined, 294
developing application servers, 306–309
division of responsibility in Shiny apps, 298–299

Shapefiles, creating choropleth maps, 248–249

Shapes, adding to Leaflet map, 264

Sharing, See Collaboration

Shiny framework
application structure, 295–299
core concepts, 294–295
designing user interfaces, 299
developing application servers, 306–309
dynamic inputs, 301–303
dynamic outputs, 303–304
e example applying to fatal police shootings, 311–318
layouts, 304–306
overview of, 293–294
publishing Shiny apps, 309–311
static content, 300–301

shinyApp(), 296–297, 299

shinyapp.io, hosting Shiny apps, 309–310

Sidebar, in Shiny example, 316

Single-bracket notation. See [] (single-bracket notation)

Slideshows, 275

snake_case
variable names, 58
writing functions, 76

Snapshots. See Commit

source(), loading and running API keys, 188

spread()
applying to educational statistics, 164–165
changing rows to columns, 158–159

Spreadsheets, working with CSV data, 124

SQL (Structured Query Language)
converting dplyr functions into SQL equivalents, 178
JOIN clause, 174–175
ORDER BY and GROUP BY clauses, 174
resources for learning, 171
SELECT statement, 171–173
WHERE clause, 173–174

SQLite
accessing from R, 176–177
SELECT statement in, 172
types of RDMSS, 169–170
WHERE clause, 173–174

ssh, accessing remote computers, 21–22

Stacked bar charts, 211–213, 239
StackOverflow, getting help via, 64
Staging area, adding files, 33. See also add (git)

Statements
   conditional, 79–80
   SQL, 171–175

Static content
   building Shiny application, 313
   Shiny framework, 300–301

Statically typed language, 60

Statistical learning
   assessing relationships, 341–342
   making predictions, 342
   overview of, 341

Statistics
   Anscombe’s Quartet, 208–209
   applying tidyR to educational statistics, 160–165
   statistical transformation of data, 237, 255

status (git)
   checking project status, 323
   checking repository status, 31–33
   pushing branches to GitHub, 329
   resolving merge conflicts, 327–328
   understanding/using git commands, 43

Strings
   character data types, 61
   rendering in R Markdown, 281–282

Style
   vs. syntax, 59

Sublime Text, selecting text editor, 7

Subplots, facets and, 244

Subset, of vector, 88–89

summarize(), tidyR core functions, 131, 138–139

Sunburst diagrams, 218, 220

Surveys, generating data, 107

SVGs (scalable vector graphics), 266

Syntax
   debugging functions, 78
   vs. style, 59

Syntax-colored code blocks, markdown options, 48

T

Tab-completion, command shells supporting, 15

Tables
   building Shiny application, 314–318
   creating data frames, 120
   data structures, 111–112
   JOIN clause, 174

markdown, 48
   referencing database table, 177
   in relational databases, 168
   rendering, 283–284

 tidyR, 157

Tagged elements, in lists, 95–96

tbl(), referencing database table, 177

Terminal (command shell). See Command line

Terminal (Linux), 5
   accessing, 9–10
   connecting to remote server, 21
   executing code, 4
   ls command, 13
   manuals (man pages), 17
   running R code, 56–57
   setting up, 4
   tab-completion support, 15

Text blocks, markdown, 46

Text editor, 6–7

Text formatting, 46

theme(), creating choropleth maps, 251

Tibble data frame, 142–143

tidyR package
   applying to educational statistics, 160–165
   changing from columns to/from rows, 157–159
   example mapping evictions in San Francisco, 252
   orienting data frames for plotting, 239
   overview of, 155–157
   reshaping data sets, 165

The tidyverse style guide
   defining variables, 58
   dplyr package, 132
   tibble data frame, 142–143
   writing functions, 76

Treemaps, 211–213, 218–220

Tutorials, for learning R, 65–66

U

UIs (user interfaces)
   application structure in Shiny framework, 295–296
   building Shiny application, 313–318
   defined, 294
   designing, 299
   division of responsibility in Shiny apps, 298–299

Unit of analysis, grouping for redefining, 144
Unordered lists, rendering Markdown lists, 282–283

URLs (Uniform Resource Identifiers)
- example finding Cuban food in Seattle, 202
- HTTP requests and, 182–184
- hyperlink syntax, 46–47

URLs (Uniform Resource Locators), 182, 286

User interfaces. See UIs (user interfaces)

Users, accessing command line, 10

Values
- creating vectors, 81–82
- modifying vectors, 92–93
- tidyr cells representing, 155
- vectors as one-dimensional collections of, 81

Variables
- anonymous, 71, 140
- breaking data into, 142
- creating intermediary variables for use in analysis, 139
- data visualization with multiple, 213–217
- data visualization with single, 210–213
- defining, 58–60
- factor variables, 126–129
- storing Shiny layouts in, 305
- tidyr columns representing, 155

VCS (version control system), 28

Vectorized functions, 86–88

Vectors
- creating, 81–83
- creating data frames, 120
- example adding, 85–86
- filtering, 90–91
- lists and, 95
- modifying, 92–93
- multiple indices, 89–90
- overview of, 81
- performing operations on, 83–84
- recycling operation, 84–85
- subsets of, 88–89
- vectorized functions, 86–88

Verbs
- dplyr package, 131
- HTTP verbs, 188–189

Version control
- accessing project history, 40–42
- adding files, 32–33
- checking repository status, 31–33
- command line in, 9
- committing changes, 33–35
- creating repository, 30–31
- forking/cloning repos and, 36–38
- git for, 4, 27–29
- GitHub for, 29
- ignoring files, 42–44
- local git process, 35
- overview of, 27
- project setup and configuration, 30
- pushing/pulling repos and, 38–40
- storing projects on GitHub, 36
- tracking changes, 32, 319–320

Version control system (VCS), 28

Videos, resources for learning R, 65

Violin plots
- data visualization with multiple variables, 215
- data visualization with single variable, 210

Visual channels, aesthetic mappings and, 237

Visual storytelling with D3, 343

Visualization. See Data visualization

VS Code (Visual Studio Code)
- preview rendering support, 49
- selecting text editor, 7

Web APIs
- access tokens (API keys), 186–188, 196–197
- accessing from R, 189–190
- example locating Cuban food in Seattle, 197–203
- flattening JSON data, 195–197
- HTTP verbs, 188–189
- overview of, 181–182
- parsing JSON data, 193–195
- processing JSON data, 191–193
- query parameters, 184–186
- RESTful requests, 182
- URLs and, 182–184

Web applications
- defined, 293
- interactive. See Shiny framework

Web browsers, Shiny framework as interface, 293–294

Web servers, 182. See also Servers

Web services. See also Web APIs
- overview of, 181
- registering with, 186–188

Webpage, URL for, 286

Websites
- creating using R Markdown, 275
publishing Shiny apps, 309–311
sharing R Markdown reports, 284–286
WHERE clause, SELECT statement, 173–174
Widgets. See Control widgets
Wildcards, command line, 17–18
Windows, icons, menus, and pointers (WIMP), 9
Windows Management Framework, 5
Windows OSs
accessing command line, 9–10
command-line tools, 4–5
installing git, 5
Windows, types of interfaces, 9

Workflows
centralized, 331
creating centralized repository, 331–333
tracking code versions with branches, 319–320
working with feature branch workflows, 333–335
working with forking workflows, 335–339

X
Xcode command line developer tools, 5

Z
Zooming, interactive data visualization, 257