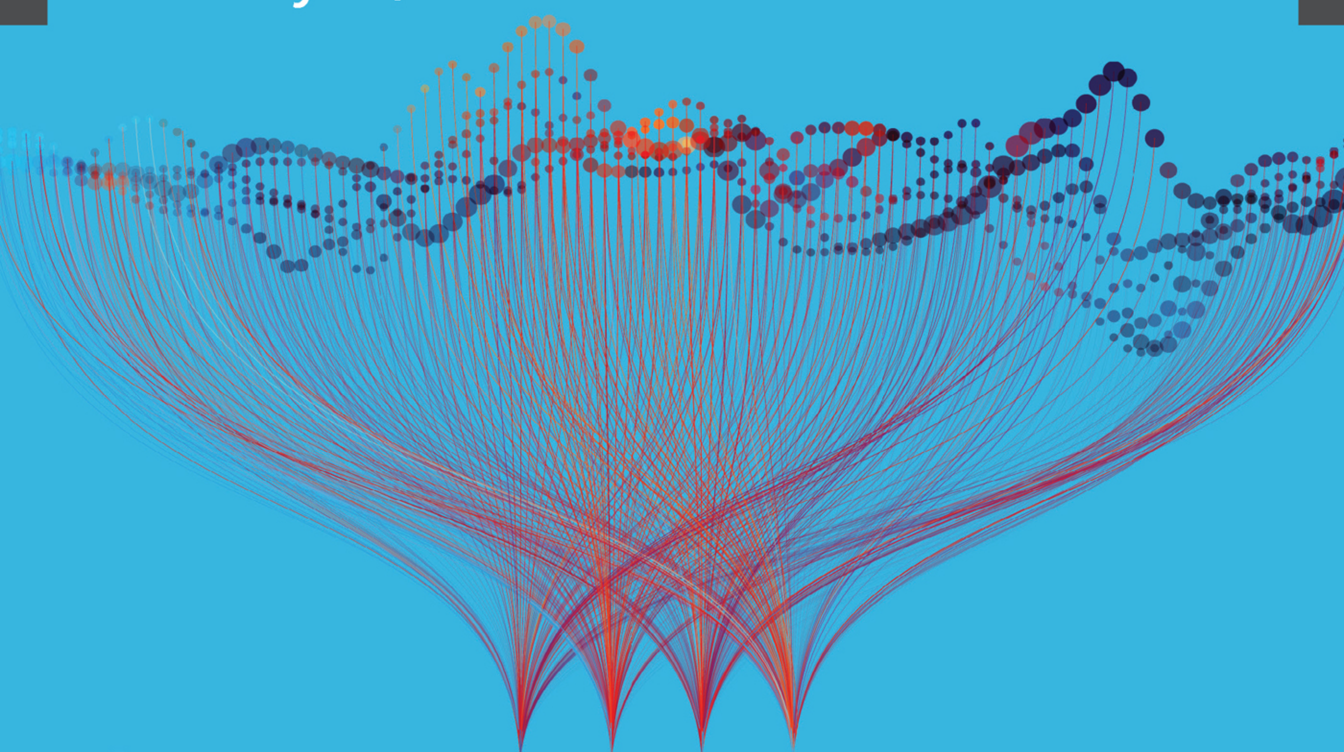
Addison Wesley Data & Analytics Series

Programming Skills for Data Science

Start Writing Code to Wrangle, Analyze, and Visualize Data with R



Michael Freeman | Joel Ross

Programming Skills for Data Science

The Pearson Addison-Wesley Data and Analytics Series

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

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.

Programming Skills for Data Science

Start Writing Code to Wrangle, Analyze, and Visualize Data with R

Michael Freeman
Joel Ross

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

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

ISBN-13: 978-0-13-513310-1
ISBN-10: 0-13-513310-6

1 18

❖

To our students who challenged us to develop better resources, and our families who supported us in the process.

❖

This page intentionally left blank

Contents

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

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:

```
# 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 website[3] for details.

Register your copy of *Programming Skills for Data Science* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135133101) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

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

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/.)

This page intentionally left blank

14

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***.

[1] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine, doctoral dissertation. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Note that this is the original specification and is very technical.

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,[2] iTunes,[3] or Reddit,[4] make (some of) their data publicly accessible through an API. This chapter will use the GitHub API[5] 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)**.[6] 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.

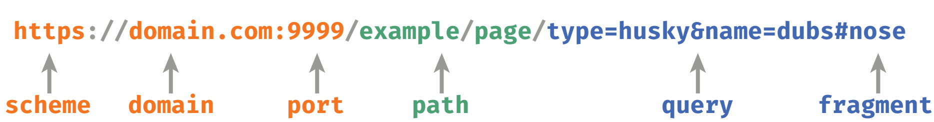


Figure 14.1 The format (schema) of a URI.

[2]**Twitter API**: https://developer.twitter.com/en/docs
[3]**iTunes search API**: https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/
[4]**Reddit API**: https://www.reddit.com/dev/api/
[5]**GitHub API**: https://developer.github.com/v3/
[6]**Uniform Resource Identifier (URI) Generic Syntax** (official technical specification): https://tools.ietf.org/html/rfc3986

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 to 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`,[7] 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

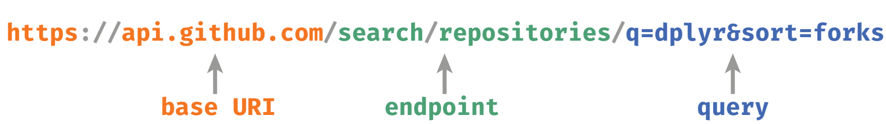


Figure 14.2 The anatomy of a web API request URI.

[7] **Nadieh 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*[a] will format the data in a somewhat more readable way. Figure 14.3 shows data formatted with this extension.
>
> [a] https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc

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.


```
{
    login: "nbremer",
    id: 5062746,
    node_id: "MDQ6VXNlcjUwNjI3NDY=",
    avatar_url: "https://avatars2.githubusercontent.com/u/5062746?v=4",
    gravatar_id: "",
    url: "https://api.github.com/users/nbremer",
    html_url: "https://github.com/nbremer",
    followers_url: "https://api.github.com/users/nbremer/followers",
    following_url: "https://api.github.com/users/nbremer/following{/other_user}",
    gists_url: "https://api.github.com/users/nbremer/gists{/gist_id}",
    starred_url: "https://api.github.com/users/nbremer/starred{/owner}{/repo}",
    subscriptions_url: "https://api.github.com/users/nbremer/subscriptions",
    organizations_url: "https://api.github.com/users/nbremer/orgs",
    repos_url: "https://api.github.com/users/nbremer/repos",
    events_url: "https://api.github.com/users/nbremer/events{/privacy}",
    received_events_url: "https://api.github.com/users/nbremer/received_events",
    type: "User",
    site_admin: false,
    name: "Nadieh Bremer",
    company: "Visual Cinnamon",
    blog: "http://www.visualcinnamon.com",
    location: "Amsterdam",
    email: null,
    hireable: null,
    bio: "Freelancing Data Visualization Designer & Artist",
    public_repos: 39,
    public_gists: 99,
    followers: 491,
    following: 0,
    created_at: "2013-07-22T07:06:53Z",
    updated_at: "2018-05-19T08:55:50Z"
}
```


Figure 14.3 GitHub API response returned by the URI `https://api.github.com/users/nbremer`, as displayed in a web browser.

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
```

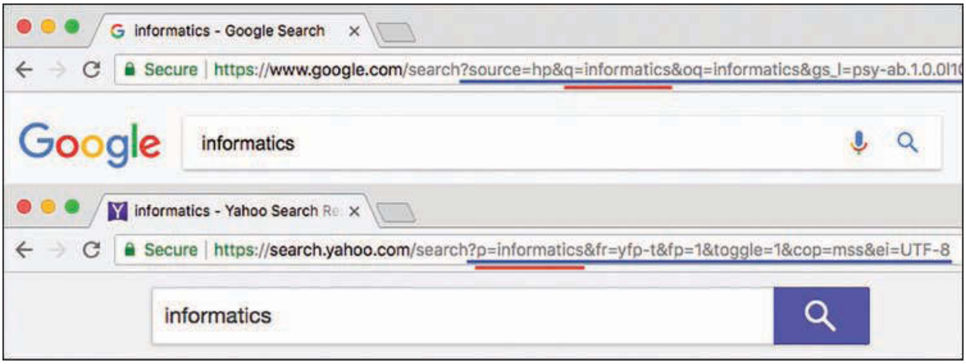


Figure 14.4 Search engine URLs for Google (top) and Yahoo (bottom) with query parameters (underlined in blue). The "search term" parameter for each web service is underlined in red.

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.

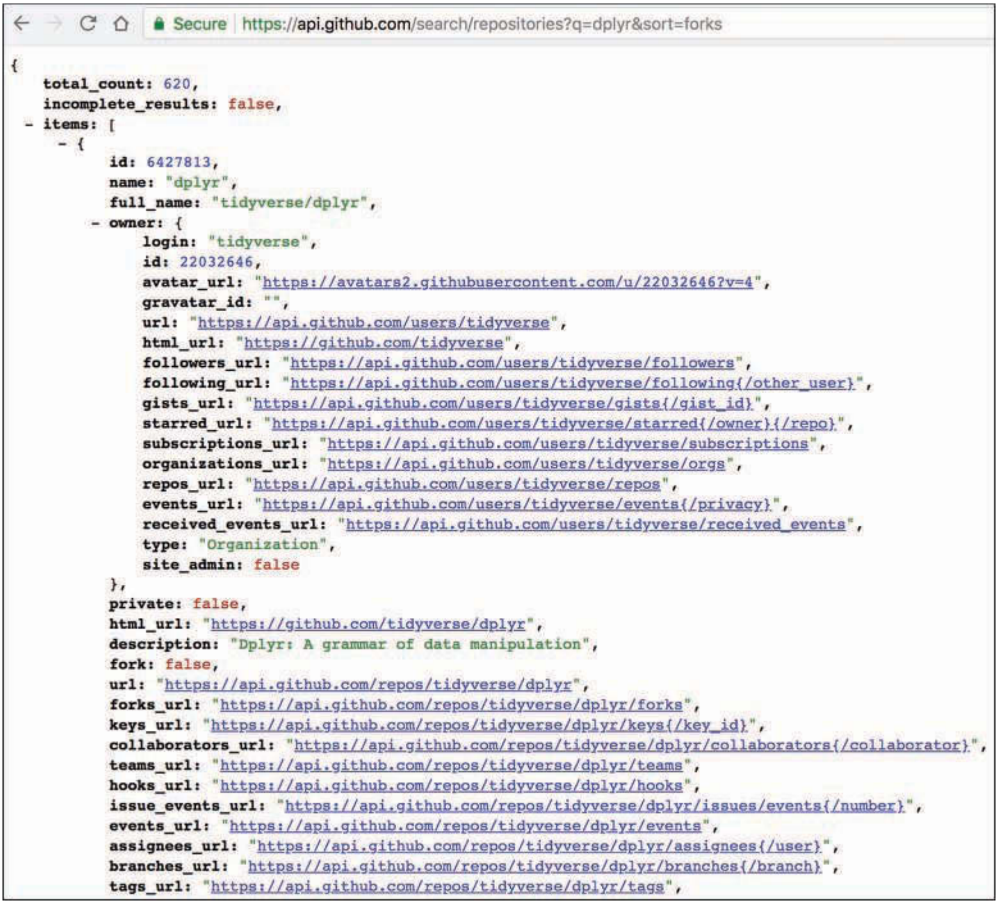
14.2.1.2 Access Tokens and API Keys

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[8]—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.

[8]**GitHub API, delete a repository** https://developer.github.com/v3/repos/#delete-a- repository




```
{
  total_count: 620,
  incomplete_results: false,
- items: [
  - {
      id: 6427813,
      name: "dplyr",
      full_name: "tidyverse/dplyr",
    - owner: {
        login: "tidyverse",
        id: 22032646,
        avatar_url: "https://avatars2.githubusercontent.com/u/22032646?v=4",
        gravatar_id: "",
        url: "https://api.github.com/users/tidyverse",
        html_url: "https://github.com/tidyverse",
        followers_url: "https://api.github.com/users/tidyverse/followers",
        following_url: "https://api.github.com/users/tidyverse/following{/other_user}",
        gists_url: "https://api.github.com/users/tidyverse/gists{/gist_id}",
        starred_url: "https://api.github.com/users/tidyverse/starred{/owner}{/repo}",
        subscriptions_url: "https://api.github.com/users/tidyverse/subscriptions",
        organizations_url: "https://api.github.com/users/tidyverse/orgs",
        repos_url: "https://api.github.com/users/tidyverse/repos",
        events_url: "https://api.github.com/users/tidyverse/events{/privacy}",
        received_events_url: "https://api.github.com/users/tidyverse/received_events",
        type: "Organization",
        site_admin: false
      },
      private: false,
      html_url: "https://github.com/tidyverse/dplyr",
      description: "Dplyr: A grammar of data manipulation",
      fork: false,
      url: "https://api.github.com/repos/tidyverse/dplyr",
      forks_url: "https://api.github.com/repos/tidyverse/dplyr/forks",
      keys_url: "https://api.github.com/repos/tidyverse/dplyr/keys{/key_id}",
      collaborators_url: "https://api.github.com/repos/tidyverse/dplyr/collaborators{/collaborator}",
      teams_url: "https://api.github.com/repos/tidyverse/dplyr/teams",
      hooks_url: "https://api.github.com/repos/tidyverse/dplyr/hooks",
      issue_events_url: "https://api.github.com/repos/tidyverse/dplyr/issues/events{/number}",
      events_url: "https://api.github.com/repos/tidyverse/dplyr/events",
      assignees_url: "https://api.github.com/repos/tidyverse/dplyr/assignees{/user}",
      branches_url: "https://api.github.com/repos/tidyverse/dplyr/branches{/branch}",
      tags_url: "https://api.github.com/repos/tidyverse/dplyr/tags",
```


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[a] for details). You can do this in R, but may want to avoid this challenge while learning how to use APIs.
>
> [a]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:

```
# 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:

```
# 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** (*REpresentational State Transfer*). 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):

```
# 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`**[9] package. As with `dplyr`, you will need to install and load this package to use it:

```
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:

```
# 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`

[9] **Getting started with httr**: official quickstart guide for `httr`: 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):

```
# 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:

```
# 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:

```r
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:

```
{
  "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[10]:

```
[
  {"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}
]
```

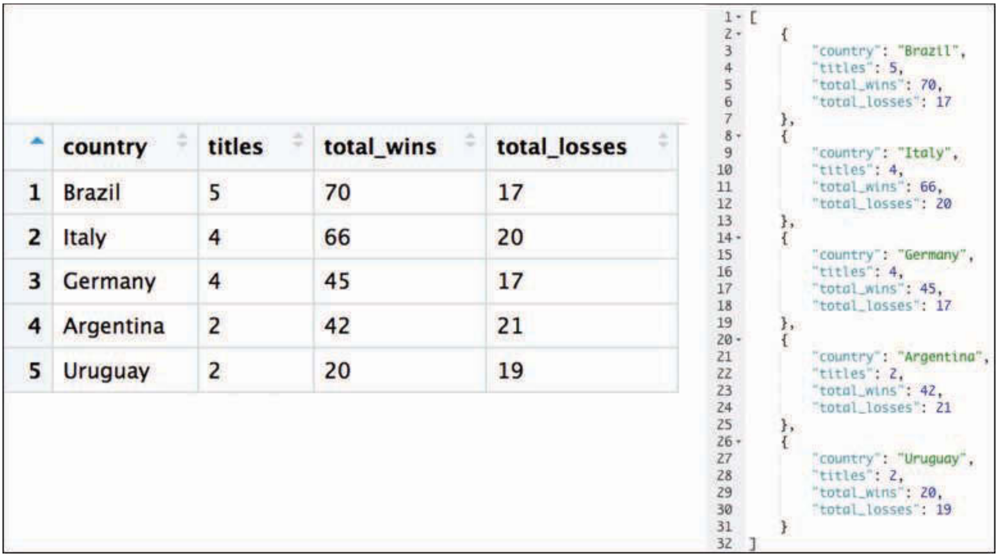
You could think of this information as a *list of lists* in R:

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



	country	titles	total_wins	total_losses
1	Brazil	5	70	17
2	Italy	4	66	20
3	Germany	4	45	17
4	Argentina	2	42	21
5	Uruguay	2	20	19

```
[
    {
        "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
    }
]
```

Figure 14.6 A data frame representation of World Cup statistics (left), which can also be represented as JSON data (right).

14.4.1 Parsing JSON

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.[11] 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!

[11] Package **`jsonlite`**: full documentation for `jsonlite`: https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf

```r
# 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.

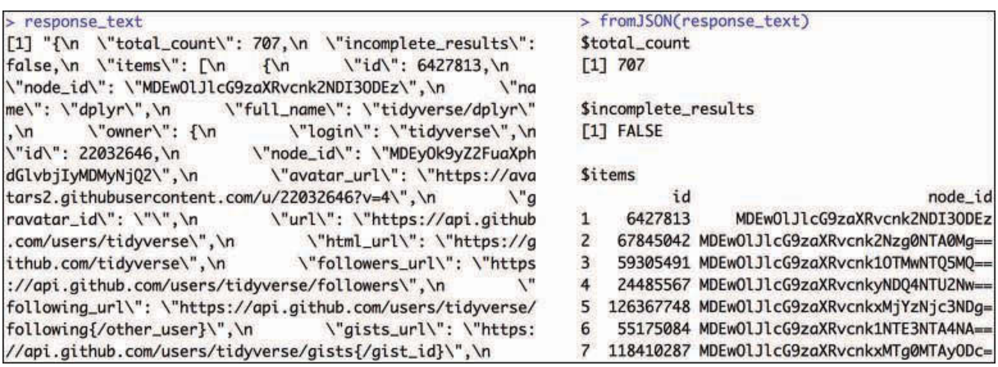


Figure 14.7 Parsing the text of an API response using `fromJSON()`. The untransformed text is shown on the left (`response_text`), which is transformed into a list (on the right) using the `fromJSON()` function.

As an example continuing the previous code:

```
# 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 *"dplry"*—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:

```
# 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"))
```

	id	node_id	name	full_name
1	6427813	MDEwOlJlcG9zaXRvcnk2NDI3ODEz	dplyr	tidyverse/dplyr
2	67845042	MDEwOlJlcG9zaXRvcnk2Nzg0NTA0Mg==	m9-dplyr	INFO-201/m9-dplyr
3	59305491	MDEwOlJlcG9zaXRvcnk1OTMwNTQ5MQ==	sparklyr	rstudio/sparklyr
4	24485567	MDEwOlJlcG9zaXRvcnkyNDQ4NTU2Nw==	dplyr-tutorial	justmarkham/dplyr-tutorial
5	126367748	MDEwOlJlcG9zaXRvcnkxMjYzNjc3NDg=	ch10-dplyr	info201/ch10-dplyr
6	55175084	MDEwOlJlcG9zaXRvcnk1NTE3NTA4NA==	tidytext	juliasilge/tidytext
7	118410287	MDEwOlJlcG9zaXRvcnkxMTg0MTAyODc=	ch10-dplyr	info201a-w18/ch10-dplyr
8	50487685	MDEwOlJlcG9zaXRvcnk1MDQ4NzY4NQ==	lecture-8-exercises	INFO-498F/lecture-8-exercises
9	86504302	MDEwOlJlcG9zaXRvcnk4NjUwNDMwMg==	dbplyr	tidyverse/dbplyr
10	84520584	MDEwOlJlcG9zaXRvcnk4NDUyMDU4NA==	Data-Analysis-with-R	susanli2016/Data-Analysis-with-R

Figure 14.8 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`).

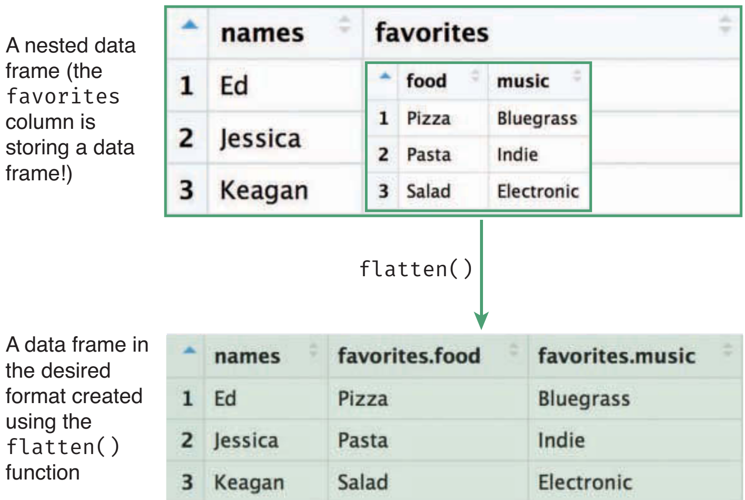


Figure 14.9 The `flatten()` function transforming a nested data frame (top) into a usable format (bottom).

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:

```
# 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

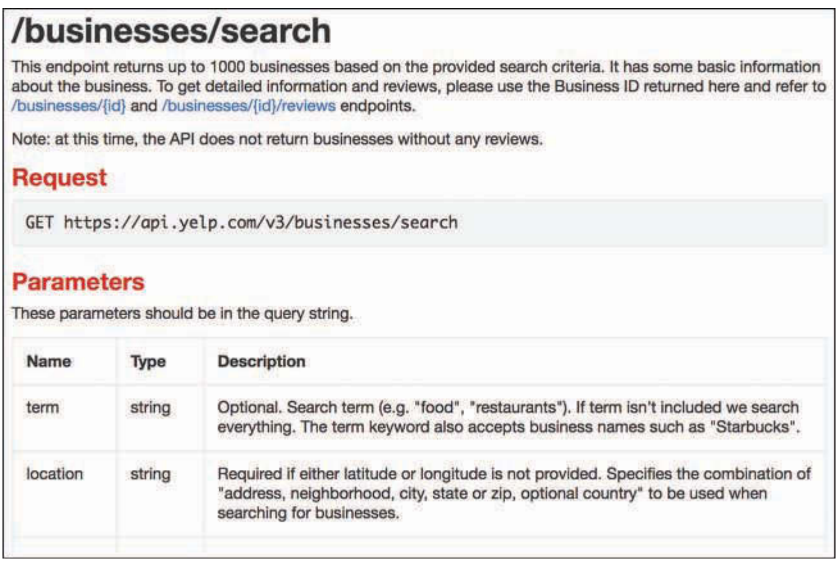
you will need to include an "Authorization" header containing your API key (in the format expected by the API) for the request to be accepted:

```
# 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,[14] a section of which is shown in Figure 14.10.



/businesses/search

This endpoint returns up to 1000 businesses based on the provided search criteria. It has some basic information about the business. To get detailed information and reviews, please use the Business ID returned here and refer to /businesses/{id} and /businesses/{id}/reviews endpoints.

Note: at this time, the API does not return businesses without any reviews.

Request

```
GET https://api.yelp.com/v3/businesses/search
```

Parameters

These parameters should be in the query string.

Name	Type	Description
term	string	Optional. Search term (e.g. "food", "restaurants"). If term isn't included we search everything. The term keyword also accepts business names such as "Starbucks".
location	string	Required if either latitude or longitude is not provided. Specifies the combination of "address, neighborhood, city, state or zip, optional country" to be used when searching for businesses.

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

[14] **Yelp Fusion API Business Search** endpoint documentation: https://www.yelp.com/developers/documentation/v3/business_search

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:

```
# 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.

```
# 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:

```
# 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.

	id	alias	name	image_url	is_closed	url
1	Wk9f5Zpnu4T6Vzf6CF5iuA	paseo-caribbean-food-fremont-seattle-2	Paseo Caribbean Food - Fremont	https://s3-media3....	FALSE	https://www.yelp.com/biz...
2	Gn5erxCRML47GgbGYdxzFA	bongos-seattle	Bongos	https://s3-media2....	FALSE	https://www.yelp.com/biz...
3	sjq3-ILJ-QYoHNejt62mYw	geos-cuban-and-creole-cafe-seattle	Geo's Cuban & Creole Cafe	https://s3-media4....	FALSE	https://www.yelp.com/biz...
4	G4j9EqGHRg2TdQVD3wE8EA	el-diablo-coffee-seattle-2	El Diablo Coffee	https://s3-media1....	FALSE	https://www.yelp.com/biz...
5	OJlYzcWkdrHhDyzwJi3bIQ	mojito-seattle	Mojito	https://s3-media2....	FALSE	https://www.yelp.com/biz...
6	XXO8vKCSqB0cz0rVTg1BJg	un-bien-seattle-seattle	Un Bien - Seattle	https://s3-media2....	FALSE	https://www.yelp.com/biz...
7	o2BJ-GAetKKTJ8yqz4YI_Q	snout-and-co-seattle-2	Snout & Co.	https://s3-media3....	FALSE	https://www.yelp.com/biz...
8	ZHErhyY2p1xd7vcuTXvbwA	cafe-con-leche-seattle	Cafe Con Leche	https://s3-media2....	FALSE	https://www.yelp.com/biz...
9	rGWsX_7SDtgPYXF6subJ3w	paseo-caribbean-food-seattle-8	Paseo Caribbean Food	https://s3-media1....	FALSE	https://www.yelp.com/biz...

Figure 14.11 A subset of the data returned by a request to the Yelp Fusion API for Cuban food in Seattle.

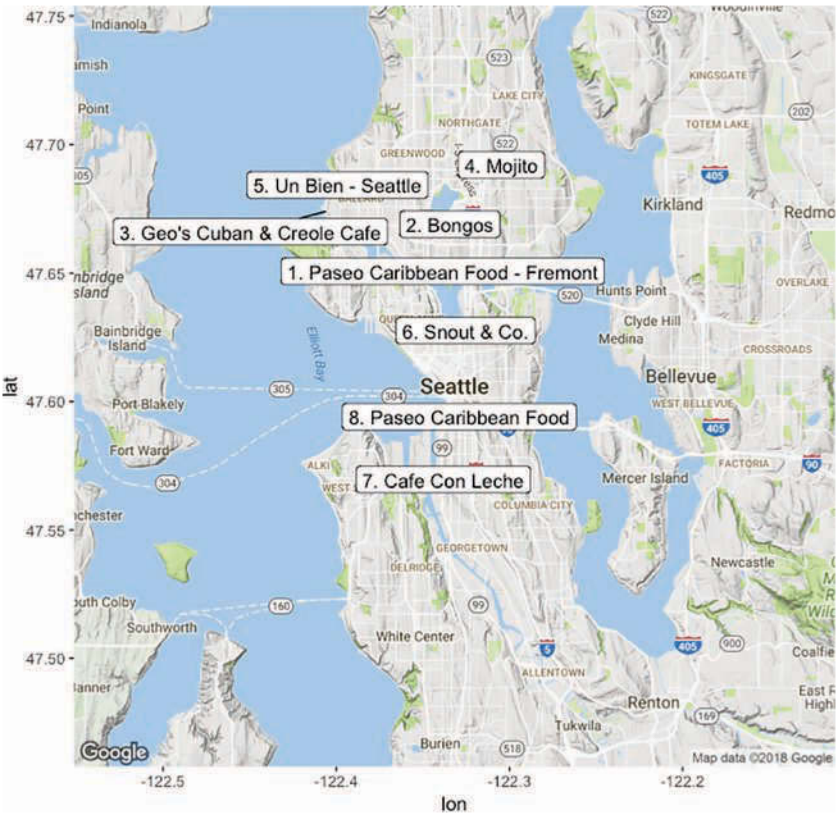


Figure 14.12 A map of the best Cuban restaurants in Seattle, according to the Yelp Fusion API.

```
# 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!

```r
# 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
restaurants <- flatten(response_data$businesses)

# Modify the data frame for analysis and presentation
restaurants <- restaurants %>%
  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]

[16] **API exercises:** https://github.com/programming-for-data-science/chapter-14-exercises

This page intentionally left blank

Index

Symbols

E

F

G

H

N

O

P

Q

R

X

Z