A beginner's guide to R and RStudio

Beyond Spreadsheets with R

Dr. Jonathan Carroll

Sample Chapter

MANNING

Beyond Spreadsheets with R

by Dr. Jonathan Carroll

Chapter 1

brief contents

1 Introducing data and the R language

This chapter covers

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

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

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

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

1.1 Data: What, where, how?

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

1.1.1 What is data?

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

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

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

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

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

1.1.2 Seeing the world as data sources

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

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

We go a step further and create more tools to summarize that data—car instrument panels to simplify the internal measurements of the engine; weather stations to summarize temperature, wind, and pressure. With the digital age, we now have an overload of data sources at our disposal. The internet provides data on virtually any and all aspects of the world we might be interested in, and we create more tools to manage these—weather, finance, social media, the number of astronauts currently in space (www.howmanypeopleareinspacerightnow.com), lists of episodes of *The Simpsons*, all available at our disposal. The world is truly made up of data.

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

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

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

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

At every point along the way, various data can be collected (anonymously, where required) and saved for later analysis. How many people requested rides to the airport this month? What was the average distance travelled? What was the average wait time?

Do people request more expensive trips from Apple or Android devices? Some of this was available previously, but it has never been easier to aggregate and compare.

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

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

1.1.3 Data munging

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

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

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

1.1.4 What you can do with well-handled data

I hope it's clear by this point that data is potentially of great importance. It is routinely more than just numbers in a table. Medical data often represents real human lives and

the effect a particular intervention has had, be that lifesavingly positive or tragically negative. These effects aren't always immediately obvious to someone viewing them from a given perspective, so it's the role of the data analyst (professional or incidental) to extract patterns from data in order to make a decision.

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

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

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

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

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

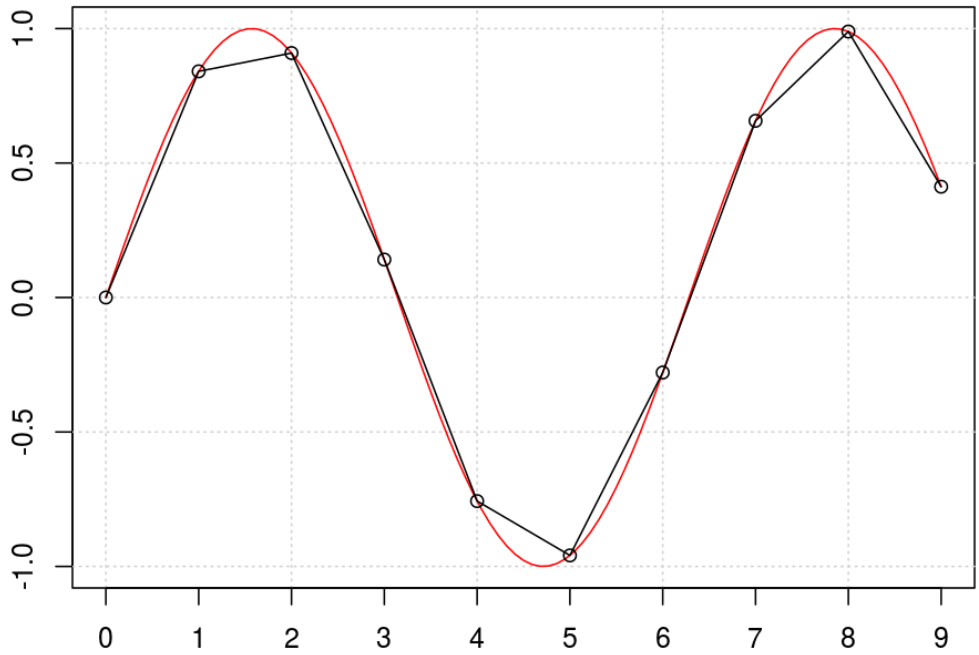

Figure 1.1 A pattern emerges. These points were generated with a `sin()` function at the values `0`, `1`, ..., `9`. The smooth `sin()` function is also plotted here.

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

Street. Once the pump was disabled, cases of cholera diminished significantly. In this case, the data was in plain sight—the locations of cholera cases—but the pattern and connection weren't immediately apparent. See figure 1.2.

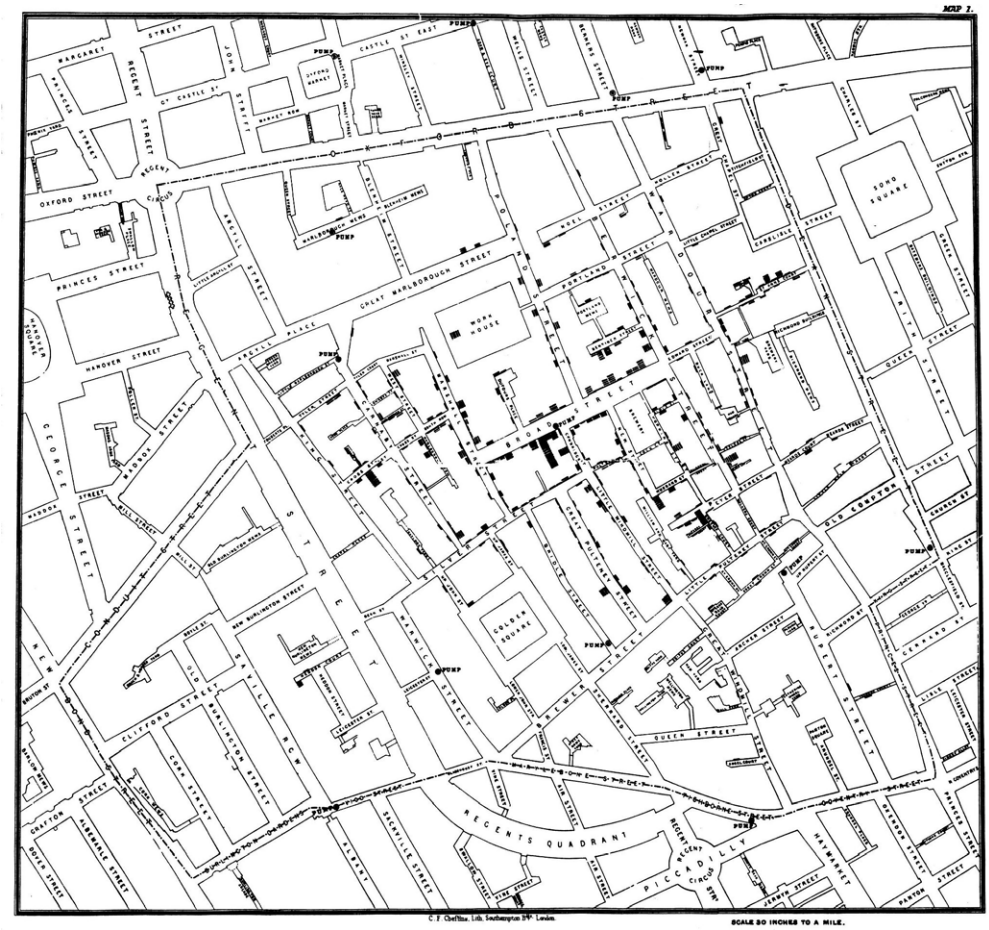

Figure 1.2 The Broad Street Cholera Map, by John Snow (public domain), via Wikimedia Commons. Dots indicate pump locations, and cases of cholera are marked with stacked bars along streets.

Perhaps unsurprisingly, several R packages are available to interact with this data. The raw data can be found in the `HistData` package, and a further graphical analysis in the `cholera` package, resulting in figure 1.3.

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

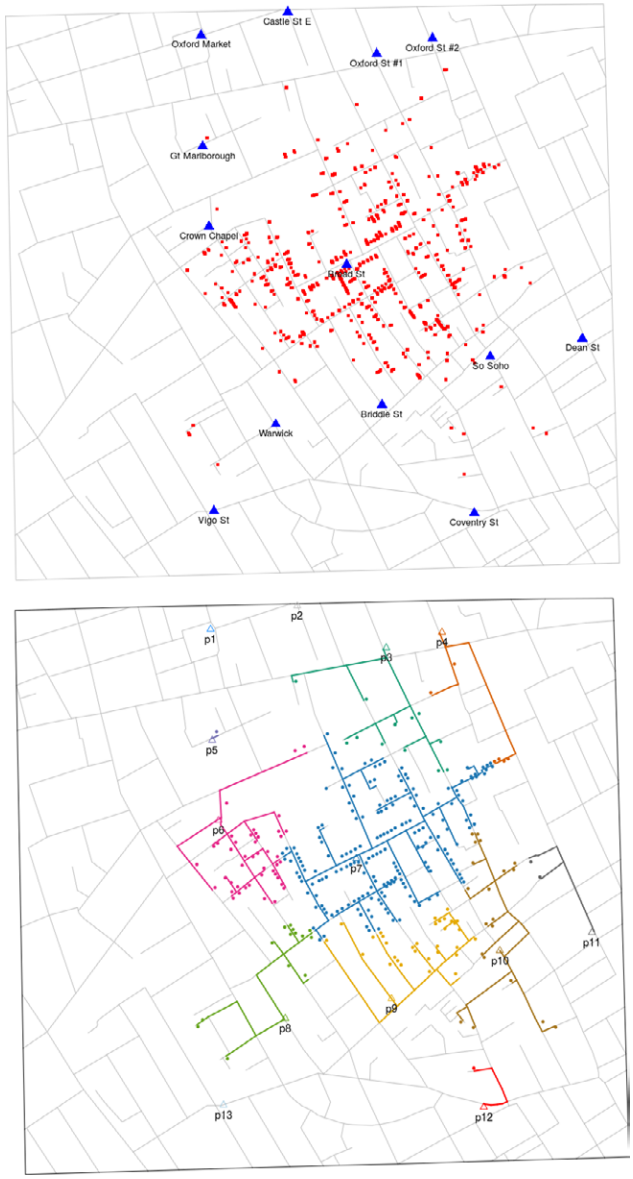

Figure 1.3 Further analyses of the Broad Street cholera data produced using the `HistData` (top) and `cholera` (bottom) R packages

1.1.5 Data as an asset

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

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

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

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

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

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

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

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

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

It's critical that any analysis you perform starts with the *right* data, data that's collected in an appropriate manner and that addresses the question you're asking.

That question needs to be the *right* one too; otherwise you won't learn what you're hoping to.

> *Far better an approximate answer to the right question, which is often vague, than the exact answer to the wrong question, which can always be made precise.*
>
> —JOHN TUKEY, founding chairman of the Princeton Statistics Department

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

1.1.6 Reproducible research and version control

Have you ever received a file with a filename like mydata_final_Thurs20May_phil_fixed_final_v2.xlsx? Not the most succinct name, but it hints at something much worse—that multiple copies of the file are floating around, each with a different version of the data, most of which is out of date due to some corrections or updates, and with unknown changes between versions. If someone presented a graph produced from one of these files, could you be certain which version it came from? Or if presented with the most recent graph and the one that preceded it, could you tell what had changed?

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

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

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

- *Plain-text file*—A file that stores its contents as numbers, letters, and punctuation, and as such can be opened in a text editor. Information in a plain-text file can be read into any system, and because it has no formatting, there's no ambiguity about what each symbol represents or how to read it. This doesn't preclude storage of formatting, but that too needs to be in plain text, such as a markup language that uses tags around values like `<b>bold text</b>`, or a markdown language that uses inline modifiers like `**bold**`.
- *Binary file*—A file that stores its contents in binary (zeroes and ones) to be interpreted by suitable software. It's not readable in a text editor but has the advantage that it can encode the formatting of data, including a variety of different formats including sounds, images, or video.

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

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

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

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

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

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

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

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

The person who follows reproducible research does the following:

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

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

With our data at the ready, our questions screaming for answers, and our intentions focused on reproducible research through version control, the only thing we still

require is a way to bring it all together to produce some results: the R programming language.

1.2 Introducing R

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

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

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

Figure 1.4 Professional users of R (rstudio.com)

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

Many other companies use R as part of their data-processing capabilities. Some well-known professional users and their specific uses include the following:[2]

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

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

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

1.2.1 The origins of R

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

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

[2] See Deepanshu Bhalla, "List of Companies Using R," *Data Science Central*, http://mng.bz/qJ66.

1.2.2 What R is and what it isn't

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

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

> *R is not a DSL. It's a language for writing DSLs, which is something that's altogether more powerful.*
>
> *One person may have a finance data goal in mind, another may be interested in natural language processing, and someone else may be aiming to predict what decisions a customer will make next. The common link between all these is data, but R is so flexible that it provides a capable mechanism to work within each of these domains.*
>
> —JOE CHENG, CTO of RStudio

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

WHAT IT IS

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

- *Open source*—The underlying source code can be freely obtained and (if desired) modified.
- *Interpreted*—R doesn't require compiling your code into a standalone program. Some languages require the code to be built into an executable in order to run it.
- *General purpose*—It isn't restricted to doing just one thing in a particular domain.
- *Functional*—It uses functions operating on unchanging data, rather than depending on the current state of the system and modifying data in place.

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

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

WHAT IT ISN'T

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

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

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

1.3 How R works

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

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

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

Being an open source language, the source for the code that runs *under the hood* is available for anyone to inspect. The official source for all versions back to R 0.60.1 is available from https://svn.r-project.org/R/branches, which means you can see how the various components of R have changed over the years if you like. That's impossible with a proprietary (closed source) program, where the internal workings are only available to those working on it. With open source software, you can even download the entire source, make changes to it, and compile your own personal version.[3]

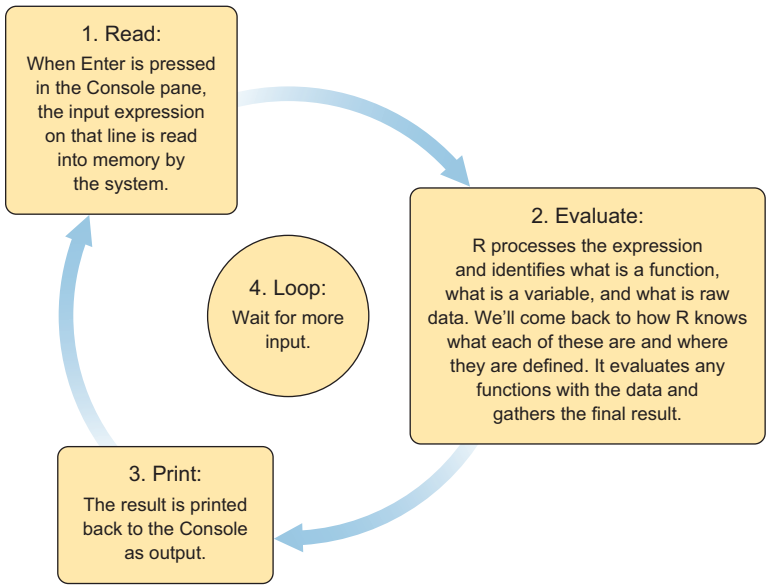

Figure 1.5 Read, evaluate, print, loop

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

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

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

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

and where to save the results. With this workflow, the analysis can be made reproducible, because armed with the raw data and the processing steps, the results can be reproduced.

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

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

On a Linux or Mac system,[5] you enter the following at the command line:

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

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

If you use R *interactively* (where you start R yourself and it produces a prompt, awaiting input), you can achieve this same behavior using the `source()` function:

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

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

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

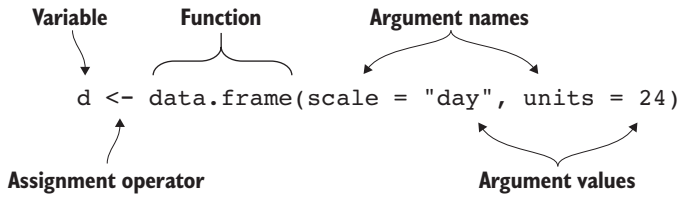

Figure 1.6 R code with some terms identified: variable, assignment operator (`<-`), function, and arguments

Some terms used in figure 1.6 may be new to you:

- *Variable*—A name to refer to a piece of data
- *Assignment operator*—Function that stores a value in a *variable*
- *Function*—Some code that interacts with data, *called* (invoked) with an opening `(` and a closing `)` (parentheses), possibly with *arguments*
- *Arguments*—Options passed to *functions*, separated by commas, possibly as pairs of argument names and argument values linked by an equals sign (`=`)—for example, `save = TRUE`

[4] Unless you set the `$PATH` environment variable to search this directory.

[5] Assuming the installation directory is in the `$PATH`.

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

1.4 Introducing RStudio

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

This can take a wide range of forms:

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

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

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

1.4.1 Working with R within RStudio

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

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

- *Editor*—This is where your scripts are written. A script is a series of commands to be executed in order. When you first open RStudio, the Editor will be empty. Click File > New File > R Script to start a new file/script.
- *Console*—The R prompt as it would appear in a terminal. This is where you enter commands line by line, followed by pressing Enter. Results returned from R are presented here.

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

- *Workspace*—The values R knows about: your data, or the *variables* you have defined appear in the Environment tab, and the history of which commands you've executed appear in the History tab.
- *Help and Plots*—Depending on which tab you have selected, Help or Plots will display either the documentation for a function or dataset, or the most recent plot produced. It also contains the Packages and Files tabs for listing installed packages and files from your computer, respectively.

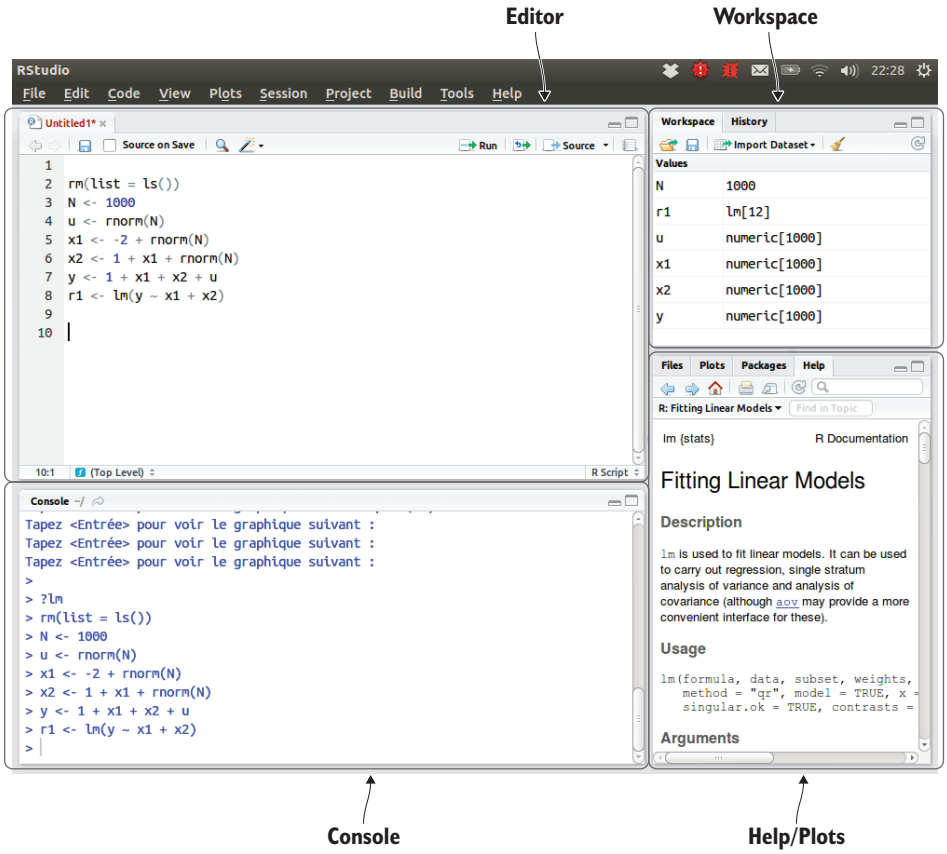

Figure 1.7 RStudio panes as they appear in Ubuntu/Linux

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

Alternatives to RStudio

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

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

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

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

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

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

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

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

>
```

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

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

Often overlooked, but these are great tips.

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

[7] Unless you deliberately set the option to start where you left off by loading the Workspace image.

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

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

TIP If you need to enter more than one command on a single line, you can separate them using a semicolon (`;`), as in `a <- 2; b <- 3`. If you begin writing a long command and are part way through—say, you have an opening `(` without a closing `)` or only half of a calculation such as `2 +`—and press Enter, R will assume you mean to insert a line break in the command and will replace the prompt with a `+` to indicate that it's waiting for the rest of the command, which you can enter as normal. R will join all the input lines together before executing them. In the Editor pane, commands can be split in the same way (over several lines) and executed either as a whole or in part. Don't worry if you get stuck in this mode. Pressing Enter again will only insert more line breaks. To exit this mode (or cancel entering a command at any time), press Esc to clear the current input and return you to the prompt (`>`).

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

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

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

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

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

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

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

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

When you want or need to start from a fresh session with nothing defined and no extra packages loaded, you can close R or RStudio and reopen it. But if you're using RStudio, a faster option is to press Ctrl-Shift-F10, which will instruct RStudio to restart the session. In some versions this will also remove all the defined objects from the Environment. If it doesn't, you can click the broom icon to clear the Workspace, as shown in figure 1.9.

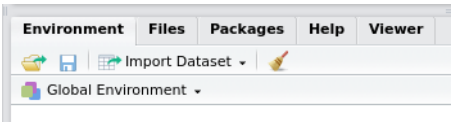

Figure 1.9 Clean up after yourself.

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

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

[8] This location depends on your operating system, but can be located from R itself as `Sys.getenv('HOME')`.

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

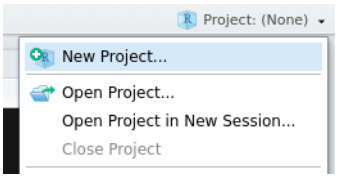

Figure 1.10 Selecting projects from the menu

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

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

1.4.2 Built-in packages (data and functions)

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

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

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

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

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

In addition to useful functions, R comes preinstalled with example datasets that are useful for demonstrations. The most commonly used ones are as follows:

- `mtcars`—Motor Trend Car Road Tests. This data, extracted from *Motor Trend* magazine in 1974, comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). This is the dataset we'll use in our examples most often. It's cliché, but clichés only get old if you've seen them too often, which in this case you probably haven't. If you need to refer to it quickly without an R session handy, a screen capture of the entire dataset is shown in the front matter.

- `iris`—A famous dataset that gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.
- `USArrests`—Violent Crime Rates by US state. This dataset contains statistics, in arrests per 100,000 residents, for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

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

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

1.4.3 Built-in documentation

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

```
?mean
```

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

```
?nonExistentFunction

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

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

```
??mean
```

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

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

1.4.4 Vignettes

A vastly underutilized (but invaluable when used correctly) feature of R is the ability to create an in-depth guide along with any package, known as a *vignette*. Vignettes can cover any topic, but generally highlight the use of a few important functions that a package provides, similar to research papers or tutorials. At several pages long, they're much more insightful than the limited information provided in the help page, which is more of a brief usage guide.

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

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

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

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

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

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

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

1.5 Try it yourself

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

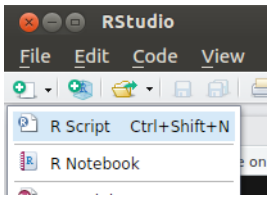

Figure 1.11 Opening a new script

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

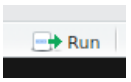

Figure 1.12 Run button

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

Terminology

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

Summary

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

R/DATA SCIENCE

Beyond Spreadsheets with R

Dr. Jonathan Carroll

Spreadsheets are powerful tools for many tasks, but if you need to interpret, interrogate, and present data, they can feel like the wrong tools for the task. That's when R programming is the way to go. The R programming language provides a comfortable environment to properly handle all types of data. And within the open source RStudio development suite, you have at your fingertips easy-to-use ways to simplify complex manipulations and create reproducible processes for analysis and reporting.

With **Beyond Spreadsheets with R** you'll learn how to go from raw data to meaningful insights using R and RStudio. Each carefully crafted chapter covers a unique way to wrangle data, from understanding individual values to interacting with complex collections of data, including data you scrape from the web. You'll build on simple programming techniques like loops and conditionals to create your own custom functions. You'll come away with a toolkit of strategies for analyzing and visualizing data of all sorts.

What's Inside

- How to start programming with R and RStudio
- Understanding and implementing important R structures and operators
- Installing and working with R packages
- Tidying, refining, and plotting your data

If you're comfortable writing formulas in Excel, you're ready for this book.

Jonathan Carroll is a data science consultant providing R programming services. He holds a PhD in theoretical physics.

"A useful guide to facilitate graduating from spreadsheets to more serious data wrangling with R."
—John D. Lewis, DDN

"An excellent book to help you understand how stored data can be used."
—Hilde Van Gysel
Trebol Engineering

"A great introduction to a data science programming language. Makes you want to learn more!"
—Jenice Tom, CVS Health

"Handy to have when your data spreads beyond a spreadsheet."
—Danil Mironov, Luxoft Poland

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/beyond-spreadsheets-with-r

ISBN-13: 978-1-61729-459-4
ISBN-10: 1-61729-459-4
54999
9 781617 294594

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]