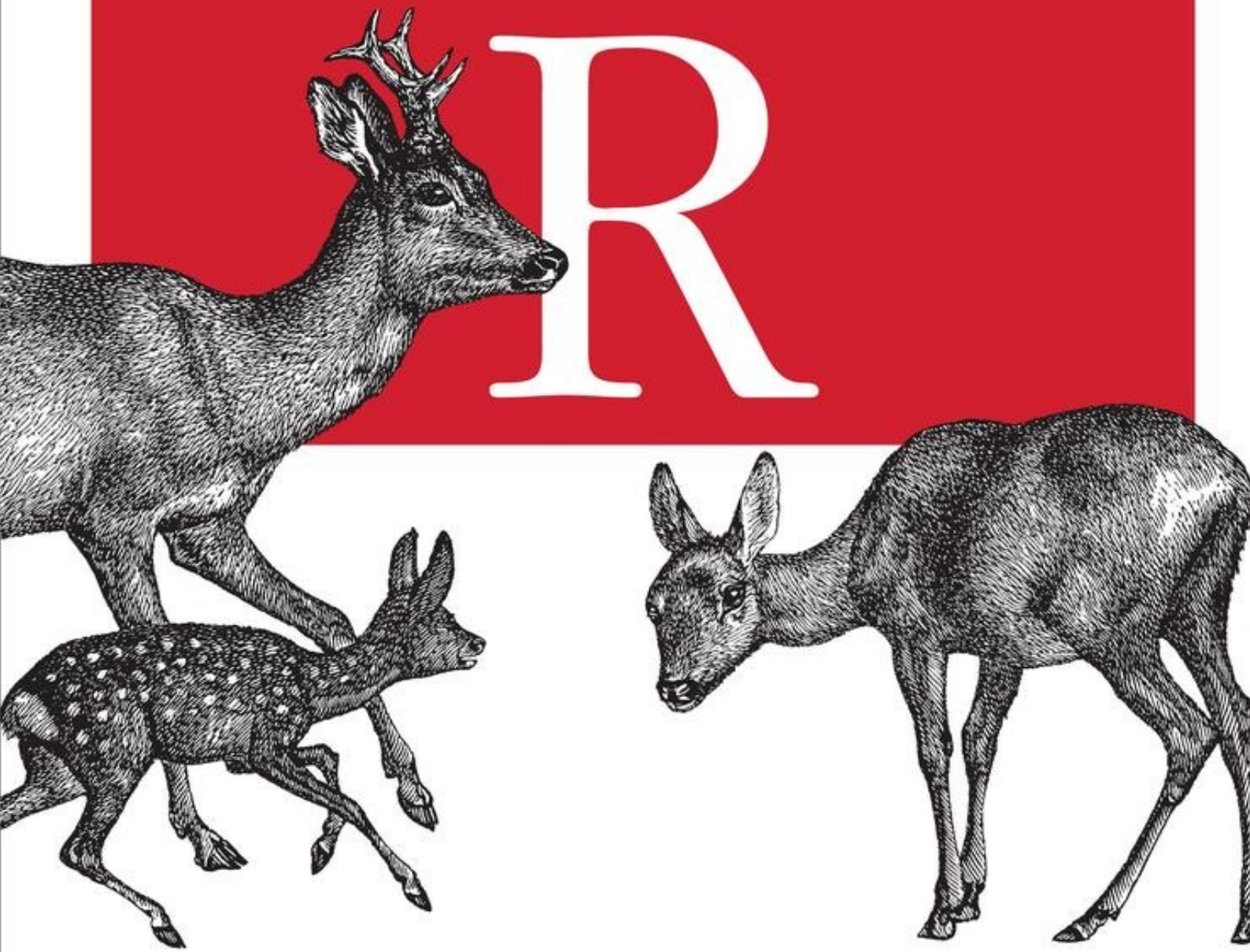
A Step-by-Step Function Guide to Data Analysis

Learning R



O'REILLY®

Richard Cotton

Learning R

Richard Cotton

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks—use your ebooks across devices without restrictions or limitations
- Multiple formats—use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing—your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of a
these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

Preface

R is a programming language and a software environment for data analysis and statistics. It is a GNU project, which means that it is free, open source software. It is growing exponentially by most measures—most estimates count over a million users, and it has over 4,000 add-on packages contributed by the community, with that number increasing by about 25% each year. The Tiobe Programming Community Index of language popularity places it at number 24 at the time of this writing, roughly on a par with SAS and MATLAB.

R is used in almost every area where statistics or data analyses are needed. Finance, marketing, pharmaceuticals, genomics, epidemiology, social sciences, and teaching are all covered, as well as dozens of other smaller domains.

About This Book

Since R is primarily designed to let you do statistical analyses, many of the books written about R focus on teaching you how to calculate statistics or model datasets. This unfortunately misses a large part of the reality of analyzing data. Unless you are doing cutting-edge research, the statistical techniques that you use will often be routine, and the modeling part of your task may not be the large one. The complete workflow for analyzing data looks more like this:

1. Retrieve some data.
2. Clean the data.
3. Explore and visualize the data.
4. Model the data and make predictions.
5. Present or publish your results.

Of course at each stage your results may generate interesting questions that lead you to look for more data, or for a different way to treat your existing data, which can send you back a step. The workflow can be iterative, but each of the steps needs to be undertaken.

The first part of this book is designed to teach you R from scratch—you don't need any experience in the language. In fact, no programming experience *at all* is necessary, but if you have some basic programming knowledge, it will help. For example, the book explains how to comment your code and how to write a `for` loop, but doesn't explain in great detail what they are. If you want a really introductory text on how to program, then *Python for Kids* by Jason R. Briggs is as good a place to start as any!

The second part of the book takes you through the complete data analysis workflow in R. Here, some basic statistical knowledge is assumed. For example, you should understand terms like *mean* and *standard deviation*, and what a bar chart is.

The book finishes with some more advanced R topics, like object-oriented programming and package creation. Garrett Grolemund's *Data Analysis with R* picks up where this book leaves off, covering data analysis workflow in more detail.

A word of warning: this isn't a reference book, and many of the topics aren't covered in great detail. This book provides tutorials to give you ideas about what you can do in R and let you practice. There isn't enough room to cover all 4,000 add-on packages, but by the time you've finished reading, you should be able to find the ones that you need, and get the help you need to start using them.

What Is in This Book

This is a book of two halves. The first half is designed to provide you with the technical skills you need to use R; each chapter is a short introduction to a different set of data types (for example, Chapter 4 covers vectors, matrices, and arrays) or a concept (for example, Chapter 8 covers branchin and looping).

The second half of the book ramps up the fun: you get to see real data analysis in action. Each chapte covers a section of the standard data analysis workflow, from importing data to publishing your results.

Here's what you'll find in Part I:

- Chapter 1, Introduction, tells you how to install R and where to get help.
- Chapter 2, A Scientific Calculator, shows you how to use R as a scientific calculator.
- Chapter 3, Inspecting Variables and Your Workspace, lets you inspect variables in different ways.
- Chapter 4, Vectors, Matrices, and Arrays, covers vectors, matrices, and arrays.
- Chapter 5, Lists and Data Frames, covers lists and data frames (for spreadsheet-like data).
- Chapter 6, Environments and Functions, covers environments and functions.
- Chapter 7, Strings and Factors, covers strings and factors (for categorical data).
- Chapter 8, Flow Control and Loops, covers branching (`if` and `else`), and basic looping.
- Chapter 9, Advanced Looping, covers advanced looping with the `apply` function and its variants.
- Chapter 10, Packages, explains how to install and use add-on packages.
- Chapter 11, Dates and Times, covers dates and times.

Here are the topics covered in Part II:

- Chapter 12, Getting Data, shows you how to import data into R.
- Chapter 13, Cleaning and Transforming, explains cleaning and manipulating data.
- Chapter 14, Exploring and Visualizing, lets you explore data by calculating statistics and plotting.
- Chapter 15, Distributions and Modeling, introduces modeling.
- Chapter 16, Programming, covers a variety of advanced programming techniques.
- Chapter 17, Making Packages, shows you how to package your work for others.

Lastly, there are useful references in Part III:

- Appendix A, contains tables comparing the properties of different types of variables.
- Appendix B, describes some other things that you can do in R.
- Appendix C, contains the answers to the end-of-chapter quizzes.
- Appendix D, contains the answers to the end of chapter programming exercises.

Which Chapters Should I Read?

If you have never used R before, then start at the beginning and work through chapter by chapter. If you already have some experience with R, you may wish to skip the first chapter and skim the chapters on the R core language.

Each chapter deals with a different topic, so although there is a small amount of dependency from on chapter to the next, it is possible to pick and choose chapters that interest you.

I recently discussed this matter with Andrie de Vries, author of *R For Dummies*. He suggested giving up and reading his book instead![1]

Conventions Used in This Book

The following font conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, file and pathnames, and file extensions.

`Constant width`
: Used for code samples that should be copied verbatim, as well as within paragraphs to refer to program elements such as variable or function names, data types, environment variables, statements, and keywords. Output from blocks of code is also in constant width, preceded by a double hash (##).

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

There is a style guide for the code used in this book at http://4dpiecharts.com/r-code-style-guide.

TIP

This icon signifies a tip, suggestion, or general note.

WARNING

This icon indicates a warning or caution.

Goals, Summaries, Quizzes, and Exercises

Each chapter begins with a list of goals to let you know what to expect in the forthcoming pages, and finishes with a summary that reiterates what you've learned. You also get a quiz, to make sure you've been concentrating (and not just pretending to read while watching telly). The answers to the questio can be found within the chapter (or at the end of the book, if you want to cheat). Finally, each chapte concludes with some exercises, most of which involve you writing some R code. After each exercise description there is a number in square brackets, denoting a generous estimate of how many minutes might take you to complete it.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at http://cran.r-project.org/web/packages/learningr.

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning R* by Richard Cotton (O'Reilly). Copyright 2013

Richard Cotton, 978-1-449-35710-8.”

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

Safari® Books Online

NOTE

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world’s leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos and prepublication manuscripts in one fully searchable database from publishers like O’Reilly Media Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Pres Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us onlin

How to Contact Us

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

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

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

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

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

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

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

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

Acknowledgments

Many amazing people have helped with the making of this book, not least my excellent editor Megha Blanchette, who is full of sensible advice.

Data was donated by several wonderful people:

- Bill Hogan of AMD found and cleaned the Alpe d'Huez cycling dataset, and pointed me toward th CDC gonorrhoea dataset. He wanted me to emphasize that he's disease-free, ladies.
- Ewan Hunter of CEFAS provided the North Sea crab dataset.
- Corina Logan of the University of Cambridge compiled and provided the deer skull data.
- Edwin Thoen of Leiden University compiled and provided the Obama vs. McCain dataset.
- Gwern Branwen compiled the hafu dataset by watching and reading an inordinate amount of manga. Kudos.

Many other people sent me datasets; there wasn't room for them all, but thank you anyway!

Bill Hogan also reviewed the book, as did Daisy Vincent of Marin Software, and JD Long. I don't know where JD works, but he lives in Bermuda, so it probably involves triangles. Additional comments and feedback were provided by James White, Ben Hanks, Beccy Smith, and Guy Bourne o TDX Group; Alex Hogg and Adrian Kelsey of HSL; Tom Hull, Karen Vanstaen, Rachel Beckett, Georgina Rimmer, Ruth Wortham, Bernardo Garcia-Carreras, and Joana Silva of CEFAS; Tal Galili Tel Aviv University; Garrett Grolemund of RStudio; and John Verzani of the City University of New York. David Maxwell of CEFAS wonderfully recruited more or less everyone else in CEFAS to revie my book.

John Verzani also deserves much credit for helping conceive this book, and for providing advice on the structure.

Sanders Kleinfeld of O'Reilly provided great tech support when I was pulling my hair out over character encodings in the manuscript. Yihui Xie went above and beyond the call of duty helping me get `knitr` to generate AsciiDoc. Rachel Head single-handedly spotted over 4,000 bugs, typos, and mistakes while copyediting.

Garib Murshudov was the lecturer who first taught me R, back in 2004.

Finally, Janette Bowler deserves a medal for her endless patience and support while I've been busy writing.

---

[1] Andrie's book covers much the same ground as *Learning R*, and in many ways is almost as good as this work, so I won't be offended if you want to read it too.

Part I. The R Language

Chapter 1. Introduction

Congratulations! You've just begun your quest to become an R programmer. So you don't pull any mental muscles, this chapter starts you off gently with a nice warm-up. Before you begin coding, we're going to talk about what R is, and how to install it and begin working with it. Then you'll try writing your first program and learn how to get help.

Chapter Goals

After reading this chapter, you should:

- Know some things that you can use R to do
- Know how to install R and an IDE to work with it
- Be able to write a simple program in R
- Know how to get help in R

What Is R?

Just to confuse you, R refers to two things. There is R, the programming language, and R, the piece o software that you use to run programs written in R. Fortunately, most of the time it should be clear from the context which R is being referred to.

R (the language) was created in the early 1990s by Ross Ihaka and Robert Gentleman, then both working at the University of Auckland. It is based upon the S language that was developed at Bell Laboratories in the 1970s, primarily by John Chambers. R (the software) is a GNU project, reflecting its status as important free and open source software. Both the language and the software are now developed by a group of (currently) 20 people known as the R Core Team.

The fact that R's history dates back to the 1970s is important, because it has evolved over the decade rather than having been designed from scratch (contrast this with, for example, Microsoft's .NET Framework, which has a much more "created"[2] feel). As with life-forms, the process of evolution ha led to some quirks and inconsistencies. The upside of the more free-form nature of R (and the free license in particular) is that if you don't like how something in R is done, you can write a package to make it do things the way that you want. Many people have already done that, and the common question now is not "Can I do this in R?" but "Which of the three implementations should I use?"

R is an interpreted language (sometimes called a scripting language), which means that your code doesn't need to be compiled before you run it. It is a high-level language in that you don't have acces to the inner workings of the computer you are running your code on; everything is pitched toward helping you analyze data.

R supports a mixture of programming paradigms. At its core, it is an imperative language (you write script that does one calculation after another), but it also supports object-oriented programming (data and functions are combined inside classes) and functional programming (functions are *first-class objects*; you treat them like any other variable, and you can call them recursively). This mix of programming styles means that R code can bear a lot of similarity to several other languages. The curly braces mean that you can write imperative code that looks like C (but the vectorized nature of R that we'll discuss in Chapter 2 means that you have fewer loops). If you use reference classes, then you can write object-oriented code that looks a bit like C# or Java. The functional programming constructs are Lisp-inspired (the variable-scoping rules are taken from the Lisp dialect, Scheme), but there are fewer brackets. All this is a roundabout way of saying that R follows the Perl ethos:

> *There is more than one way to do it.*
>
> — Larry Wall

Installing R

If you are using a Linux machine, then it is likely that your package manager will have R available, though possibly not the latest version. For everyone else, to install R you must first go to

http://www.r-project.org. Don't be deceived by the slightly archaic website;[3] it doesn't reflect on the quality of R. Click the link that says "download R" in the "Getting Started" pane at the bottom of the page.

Once you've chosen a mirror close to you, choose a link in the "Download and Install R" pane at the top of the page that's appropriate to your operating system. After that there are one or two OS-specif clicks that you need to make to get to the download.

If you are a Windows user who doesn't like clicking, there is a cheeky shortcut to the setup file at *http://<CRAN MIRROR>/bin/windows/base/release.htm.*

Choosing an IDE

If you use R under Windows or Mac OS X, then a graphical user interface (GUI) is available to you. This consists of a command-line interpreter, facilities for displaying plots and help pages, and a basi text editor. It is perfectly possible to use R in this way, but for serious coding you'll at least want to use a more powerful text editor. There are countless text editors for programmers; if you already hav a favorite, then take a look to see if you can get syntax highlighting of R code for it.

If you aren't already wedded to a particular editor, then I suggest that you'll get the best experience o R by using an integrated development environment (IDE). Using an IDE rather than a separate text editor gives you the benefit of only using one piece of software rather than two. You get all the facilities of the stock R GUI, but with a better editor, and in some cases things like integrated version control.

The following sections introduce five popular choices, but this is by no means an exhaustive list (a few additional suggestions follow). It is worth trying several IDEs; a development environment is a piece of software that you could be spending thousands of hours using, so it's worth taking the time t find one[4] that you like. A few additional suggestions follow this selection.

Emacs + ESS

Although Emacs calls itself a text editor, 36 years (and counting) of development have given it an unparalleled number of features. If you've been programming for any substantial length of time, you probably already know whether or not you want to use it. Converts swear by its limitless customizability and raw editing power; others complain that it overcomplicates things and that the ke chords give them repetitive strain injury. There is certainly a steep learning curve, so be willing to spend a month or two getting used to it. The other big benefit is that Emacs is not R-specific, so you can use it for programming in many languages. The original version of Emacs is (like R) a GNU project, available from http://www.gnu.org/software/emacs/.

Another popular fork is XEmacs, available from http://www.xemacs.org/.

Emacs Speaks Statistics (ESS) is an add-on for Emacs that assists you in writing R code. Actually, it works with S-Plus, SAS, and Stata, too, so you can write statistical code with whichever package you like (choose R!). Several of the authors of ESS are also R Core Team members, so you are guarantee good integration with R. It is available through the Emacs package management system, or you can download it from http://ess.r-project.org/.

Use it if you want to write code in multiple languages, you want the most powerful editor available, and you are fearless with learning curves.

Eclipse/Architect

Eclipse is another cross-platform IDE, widely used in the Java community. Like Emacs, it is very powerful, and its plug-in system makes it highly customizable. The learning curve is shallower, though, and it allows for more pointing and clicking than the heavily keyboard-driven Emacs.

Architect is an R-oriented variant of Eclipse developed by statistics consultancy Open Analytics. It includes the StatET plug-in for integration with R, including a debugger that is superior to the one built into R GUI. Download it from http://www.openanalytics.eu/downloads/architect.

Alternatively, you can get the standard Eclipse IDE from http://eclipse.org and use its package manager to download the StatET plug-in from http://www.walware.de/goto/statet.

Use it if you want to write code in multiple languages, you don't have time to learn Emacs, and you don't mind a several-hundred-megabyte install.

RStudio

RStudio is an R-specific IDE. That means that you lose the ability to code (easily) in multiple languages, but you do get some features especially for R. For example, the plot windows are better than the R GUI originals, and there are facilities for publishing code. The editor is more basic than either Emacs or Eclipse, but it's good enough for most purposes, and is easier to get started with than the other two. RStudio's party trick is that you can run it remotely through a browser, so you can run on a powerful server, then access it from a netbook (or smartphone) without loss of computational power. Download it from http://www.rstudio.org.

Use it if you mainly write R code, don't need advanced editor features, and want a shallow learning curve or the ability to run remotely.

Revolution-R

Revolution-R comes in two flavors: the free (as in beer) community edition and the paid-for enterpri edition. Both take a different tack from the other IDEs mentioned so far: whereas Emacs, Eclipse, an RStudio are pure graphical frontends that let you connect to any version of R, Revolution-R ships wit its own customized version of R. Typically this is a stable release, one or two versions back from the most current. It also has some enhancements for working with big data, and some enterprise-related features. Download it from http://www.revolutionanalytics.com/products/revolution-r.php.

Use it if you mainly write R code, you work with big data or want a paid support contract, or you require extra stability in your R platform.

Live-R

Live-R is a new player, in invite-only beta at the time this book is going to press. It provides an IDE for R as a web application. This avoids all the hassle of installing software on your machine and, like RStudio's remote installation, gives you the ability to run R calculations from an underpowered machine. Live-R also includes a number of features for collaboration, including a shared editor and code publishing, as well as some admin tools for running courses based upon R. The main downside is that not all the add-on packages for R are available; you are currently limited to about 200 or so that are compatible with the web application. Sign up at http://live-analytics.com/.

Use it if you mainly write R code, don't want to install any software, or want to teach a class based upon R.

Other IDEs and Editors

There are many more editors that you can use to write R code. Here's a quick roundup of a few more possibilities:

- JGR [pronounced "Jaguar"] is a Java-based GUI for R, essentially a souped-up version of the stoc R GUI.
- Tinn-R is a fork of the editor TINN that has extensions specifically to help you write R code.
- SciViews-K, from the same team that makes Tinn-R, is an extension for the Komodo IDE to work with R.
- Vim-R is a plug-in for Vim that provides R integration.
- NppToR plugs into Notepad++ to give R integration.

Your First Program

It is a law of programming books that the first example shall be a program to print the phrase "Hello world!" In R that's really boring, since you just type "Hello world!" at the command prompt, and it will parrot it back to you. Instead, we're going to write the simplest statistical program possible.

Open up R GUI, or whichever IDE you've decided to use, find the command prompt (in the code editor window), and type:

```
mean(1:5)
```

Hit Enter to run the line of code. Hopefully, you'll get the answer `3`. As you might have guessed, this code is calculating the arithmetic mean of the numbers from 1 to 5. The colon operator, `:`, creates a sequence of numbers from the first number, in this case 1, to the second number (5), each separated b 1. The resulting sequence is called a *vector*. `mean` is a *function* (that calculates the arithmetic mean), and the vector that we enclose inside the parentheses is called an *argument* to the function.

Well done! You've calculated a statistic using R.

> **TIP**
>
> In R GUI and most of the IDEs mentioned here, you can press the up arrow key to cycle back through previous commands.

How to Get Help in R

Before you get started writing R code, the most important thing to know is how to get help. There are lots of ways to do this. Firstly, if you want help on a function or a dataset that you know the name of, type `?` followed by the name of the function. To find functions, type two question marks (`??`) followe by a keyword related to the problem to search. Special characters, reserved words, and multiword search terms need enclosing in double or single quotes. For example:

```
?mean                 #opens the help page for the mean function
?"+"                  #opens the help page for addition
?"if"                 #opens the help page for if, used for branching code
??plotting            #searches for topics containing words like "plotting"
??"regression model"  #searches for topics containing phrases like this
```

NOTE

That # symbol denotes a comment. It means that R will ignore the rest of the line. Use comments to document your code, so that you can remember what you were doing six months ago.

The functions `help` and `help.search` do the same things as `?` and `??`, respectively, but with these you always need to enclose your arguments in quotes. The following commands are equivalent to the previous lot:

```
help("mean")
help("+")
help("if")
help.search("plotting")
help.search("regression model")
```

The `apropos` function[5] finds variables (including functions) that match its input. This is really usefu if you can only half-remember the name of a variable that you've created, or a function that you wan to use. For example, suppose you create a variable `a_vector`:

```
a_vector <- c(1, 3, 6, 10)
```

You can then recall this variable using `apropos`:

```
apropos("vector")
```

```
## [1] ".__C__vector"         "a_vector"             "as.data.frame.vector"
## [4] "as.vector"            "as.vector.factor"     "is.vector"
## [7] "vector"               "Vectorize"
```

The results contain the variable you just created, `a_vector`, and all other variables that contain the string `vector`. In this case, all the others are functions that are built into R.

Just finding variables that contain a particular string is fine, but you can also do fancier matching wit `apropos` using regular expressions.

NOTE

Regular expressions are a cross-language syntax for matching strings. The details will only be touched upon in this book, but you need to learn to use them; they'll change your life. Start at http://www.regular-expressions.info/quickstart.html, and then try Michael Fitzgerald's *Introducing Regular Expressions*.

A simple usage of `apropos` could, for example, find all variables that end in `z`, or to find all variables containing a number between 4 and 9:

```
apropos("z$")
```

```
## [1] "alpe_d_huez" "alpe_d_huez" "force_tz"    "indexTZ"     "SSgompertz"
## [6] "toeplitz"    "tz"          "unz"         "with_tz"
```

```
apropos("[4-9]")
```

```
##  [1] ".__C__S4"            ".__T__xmlToS4:XML"   ".parseISO8601"
##  [4] ".SQL92Keywords"      ".TAOCP1997init"      "asS4"
##  [7] "assert_is_64_bit_os" "assert_is_S4"        "base64"
## [10] "base64Decode"        "base64Encode"        "blues9"
## [13] "car90"               "enc2utf8"            "fixPre1.8"
## [16] "Harman74.cor"        "intToUtf8"           "is_64_bit_os"
```

```
## [19] "is_S4"              "isS4"              "seemsS4Object"
## [22] "state.x77"          "to.minutes15"      "to.minutes5"
## [25] "utf8ToInt"          "xmlToS4"
```

Most functions have examples that you can run to get a better idea of how they work. Use the `exampl` function to run these. There are also some longer demonstrations of concepts that are accessible with the `demo` function:

```
example(plot)
demo()         #list all demonstrations
demo(Japanese)
```

R is modular and is split into *packages* (more on this later), some of which contain *vignettes*, which are short documents on how to use the packages. You can browse all the vignettes on your machine using `browseVignettes`:

```
browseVignettes()
```

You can also access a specific vignette using the `vignette` function (but if your memory is as bad as mine, using `browseVignettes` combined with a page search is easier than trying to remember the name of a vignette and which package it's in):

```
vignette("Sweave", package = "utils")
```

The help search operator `??` and `browseVignettes` will only find things in packages that you have installed on your machine. If you want to look in *any* package, you can use `RSiteSearch`, which runs query at http://search.r-project.org. Multiword terms need to be wrapped in braces:

```
RSiteSearch("{Bayesian regression}")
```

TIP

Learning to help yourself is extremely important. Think of a keyword related to your work and try `?`, `??`, `apropos`, and `RSiteSearch` with it.

There are also lots of R-related resources on the Internet that are worth trying. There are too many to list here, but start with these:

- R has a number of mailing lists with archives containing years' worth of questions on the languag At the very least, it is worth signing up to the general-purpose list, *R-help*.
- RSeek is a web search engine for R that returns functions, posts from the R mailing list archives, and blog posts.
- R-bloggers is the main R blogging community, and the best way to stay up to date with news and tips about R.
- The programming question and answer site Stack Overflow also has a vibrant R community, providing an alternative to the *R-help* mailing list. You also get points and badges for answering questions!

Installing Extra Related Software

There are a few other bits of software that R can use to extend its functionality. Under Linux, your package manager should be able to retrieve them. Under Windows, rather than hunting all over the Internet to track down this software, you can use the `installr` add-on package to automatically insta

these extra pieces of software. None of this software is compulsory, so you can skip this section now you want, but it's worth knowing that the package exists when you come to need the additional software. Installing and loading packages is discussed in detail in Chapter 10, so don't worry if you don't understand the commands yet:

```
install.packages("installr")   #download and install the package named installr
library(installr)              #load the installr package
install.RStudio()              #download and install the RStudio IDE
install.Rtools()               #Rtools is needed for building your own packages
install.git()                  #git provides version control for your code
```

Summary

- R is a free, open source language for data analysis.
- It's also a piece of software used to run programs written in R.
- You can download R from http://www.r-project.org.
- You can write R code in any text editor, but there are several IDEs that make development easier.
- You can get help on a function by typing `?` then its name.
- You can find useful functions by typing `??` then a search string, or by calling the `apropos` functio
- There are many online resources for R.

Test Your Knowledge: Quiz

Question 1-1
: Which language is R an open source version of?

Question 1-2
: Name at least two programming paradigms in which you can write R code.

Question 1-3
: What is the command to create a vector of the numbers from 8 to 27?

Question 1-4
: What is the name of the function used to search for help within R?

Question 1-5
: What is the name of the function used to search for R-related help on the Internet?

Test Your Knowledge: Exercises

Exercise 1-1
: Visit http://www.r-project.org, download R, and install it. For extra credit, download and install one of the IDEs mentioned in Other IDEs and Editors. [30]

Exercise 1-2
: The function `sd` calculates the standard deviation. Calculate the standard deviation of the number from 0 to 100. Hint: the answer should be about `29.3`. [5]

Exercise 1-3
: Watch the demonstration on mathematical symbols in plots, using `demo(plotmath)`. [5]

---

[2] Intelligently designed?

[3] A look in the Internet Archive's Wayback Machine suggests that the front page hasn't changed much since May 2004.

[4] You don't need to limit yourself to just *one* way of using R. I have IDE commitment issues and use a mix of Eclipse + StatET, RStudio, Live-R, Tinn-R, Notepad++, and R GUI. Experiment, and find something that works for you.

[5] `apropos` is Latin for "A Unix program that finds manpages."

Chapter 2. A Scientific Calculator

R is at heart a supercharged scientific calculator, so it has a fairly comprehensive set of mathematica capabilities built in. This chapter will take you through the arithmetic operators, common mathematical functions, and relational operators, and show you how to assign a value to a variable.

Chapter Goals

After reading this chapter, you should:

- Be able to use R as a scientific calculator
- Be able to assign a variable and view its value
- Be able to use infinite and missing values
- Understand what logical vectors are and how to manipulate them

Mathematical Operations and Vectors

The + operator performs addition, but it has a special trick: as well as adding two numbers together, you can use it to add two vectors. A *vector* is an ordered set of values. Vectors are tremendously important in statistics, since you will usually want to analyze a whole dataset rather than just one piece of data.

The colon operator, `:`, which you have seen already, creates a sequence from one number to the next, and the `c` function concatenates values, in this case to create vectors (*concatenate* is a Latin word meaning "connect together in a chain").

Variable names are case sensitive in R, so we need to be a bit careful in this next example. The `C` function does something completely different to `c`:[6]

```
1:5 + 6:10          #look, no loops!
```

```
## [1]  7  9 11 13 15
```

```
c(1, 3, 6, 10, 15) + c(0, 1, 3, 6, 10)
```

```
## [1]  1  4  9 16 25
```

TIP

The colon operator and the `c` function are used almost everywhere in R code, so it's good to practice using them. Try creating some vectors of your own now.

If we were writing in a language like C or Fortran, we would need to write a loop to perform addition on all the elements in these vectors. The vectorized nature of R's addition makes things easy, letting us avoid the loop. Vectors will be discussed more in Logical Vectors.

Vectorized has several meanings in R, the most common of which is that an operator or a function wi act on each element of a vector without the need for you to explicitly write a loop. (This built-in implicit looping over elements is also much faster than explicitly writing your own loop.) A second meaning of vectorization is when a function takes a vector as an input and calculates a summary statistic:

```
sum(1:5)
```

```
## [1] 15
```

```
median(1:5)
```

```
## [1] 3
```

A third, much less common case of vectorization is *vectorization over arguments*. This is when a function calculates a summary statistic from several of its input arguments. The `sum` function does

this, but it is very unusual. median does not:

```
sum(1, 2, 3, 4, 5)
```

```
## [1] 15
```

```
median(1, 2, 3, 4, 5)  #this throws an error
```

```
## Error: unused arguments (3, 4, 5)
```

All the arithmetic operators in R, not just plus (+), are vectorized. The following examples demonstrate subtraction, multiplication, exponentiation, and two kinds of division, as well as remainder after division:

```
c(2, 3, 5, 7, 11, 13) - 2           #subtraction
```

```
## [1]  0  1  3  5  9 11
```

```
-2:2 * -2:2                         #multiplication
```

```
## [1] 4 1 0 1 4
```

```
identical(2 ^ 3, 2 ** 3)            #we can use ^ or ** for exponentiation
                                    #though ^ is more common
```

```
## [1] TRUE
```

```
1:10 / 3                            #floating point division
```

```
##  [1] 0.3333 0.6667 1.0000 1.3333 1.6667 2.0000 2.3333 2.6667 3.0000 3.3333
```

```
1:10 %/% 3                          #integer division
```

```
##  [1] 0 0 1 1 1 2 2 2 3 3
```

```
1:10 %% 3                           #remainder after division
```

```
##  [1] 1 2 0 1 2 0 1 2 0 1
```

R also contains a wide selection of mathematical functions. You get trigonometry (sin, cos, tan, and their inverses asin, acos, and atan), logarithms and exponents (log and exp, and their variants log1 and expm1 that calculate log(1 + x) and exp(x - 1) more accurately for very small values of x), and almost any other mathematical function you can think of. The following examples provide a hint of what is on offer. Again, notice that all the functions naturally operate on vectors rather than just single values:

```
cos(c(0, pi / 4, pi / 2, pi))       #pi is a built-in constant
```

```
## [1]  1.000e+00  7.071e-01  6.123e-17 -1.000e+00
```

```
exp(pi * 1i) + 1                    #Euler's formula
```

```
## [1] 0+1.225e-16i
```

```
factorial(7) + factorial(1) - 71 ^ 2 #5041 is a great number
```

```
## [1] 0
```

```
choose(5, 0:5)
```

```
## [1]  1  5 10 10  5  1
```

To compare integer values for equality, use ==. Don't use a single = since that is used for something else, as we'll see in a moment. Just like the arithmetic operators, == and the other relational operator are vectorized. To check for inequality, the "not equals" operator is !=. Greater than and less than are

as you might expect: > and < (or >= and <= if equality is allowed). Here are a few examples:

```
c(3, 4 - 1, 1 + 1 + 1) == 3                #operators are vectorized too
```

```
## [1] TRUE TRUE TRUE
```

```
1:3 != 3:1
```

```
## [1]  TRUE FALSE  TRUE
```

```
exp(1:5) < 100
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
```

```
(1:5) ^ 2 >= 16
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

Comparing nonintegers using == is problematic. All the numbers we have dealt with so far are floatin point numbers. That means that they are stored in the form `a * 2 ^ b`, for two numbers `a` and `b`. Sin this whole form has to be stored in 32 bits, the resulting number is only an approximation of what yo really want. This means that rounding errors often creep into calculations, and the answers you expected can be wildly wrong. Whole books have been written on this subject; there is too much to worry about here. Since this is such a common mistake, the FAQ on R has an entry about it, and it's a good place to start if you want to know more.

Consider these two numbers, which should be the same:

```
sqrt(2) ^ 2 == 2          #sqrt is the square-root function
```

```
## [1] FALSE
```

```
sqrt(2) ^ 2 - 2           #this small value is the rounding error
```

```
## [1] 4.441e-16
```

R also provides the function `all.equal` for checking equality of numbers. This provides a tolerance level (by default, about `1.5e-8`), so that rounding errors less than the tolerance are ignored:

```
all.equal(sqrt(2) ^ 2, 2)
```

```
## [1] TRUE
```

If the values to be compared are not the same, `all.equal` returns a report on the differences. If you require a `TRUE` or `FALSE` value, then you need to wrap the call to `all.equal` in a call to `isTRUE`:

```
all.equal(sqrt(2) ^ 2, 3)
```

```
## [1] "Mean relative difference: 0.5"
```

```
isTRUE(all.equal(sqrt(2) ^ 2, 3))
```

```
## [1] FALSE
```

NOTE

To check that two numbers are the same, don't use ==. Instead, use the `all.equal` function.

We can also use == to compare strings. In this case the comparison is case sensitive, so the strings must match exactly. It is also theoretically possible to compare strings using greater than or less than (> and <):

```
c(
  "Can", "you", "can", "a", "can", "as",
  "a", "canner", "can", "can", "a", "can?"
) == "can"
```

```
##  [1] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [12] FALSE
```

```
c("A", "B", "C", "D") < "C"
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```
c("a", "b", "c", "d") < "C" #your results may vary
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

In practice, however, the latter approach is almost always an awful idea, since the results depend upo your locale (different cultures are full of odd sorting rules for letters; in Estonian, “z” comes betwee “s” and “t”). More powerful string matching functions will be discussed in Cleaning Strings.

TIP

The help pages `?Arithmetic`, `?Trig`, `?Special`, and `?Comparison` have more examples, and explain the gory details of what happens in edge cases. (Try `0 ^ 0` or integer division on nonintegers if you are curious.)

Assigning Variables

It’s all very well calculating things, but most of the time we want to store the results for reuse. We ca assign a (local) variable using either `<-` or `=`, though for historical reasons, `<-` is preferred:

```
x <- 1:5
y = 6:10
```

Now we can reuse these values in our further calculations:

```
x + 2 * y - 3
```

```
## [1] 10 13 16 19 22
```

Notice that we didn’t have to declare what types of variables `x` and `y` were going to be before we assigned them (unlike in most compiled languages). In fact, we *couldn’t* have declared the type, sinc no such concept exists in R.

Variable names can contain letters, numbers, dots, and underscores, but they can’t start with a numbe or a dot followed by a number (since that looks too much like a number). Reserved words like “if” an “for” are not allowed. In some locales, non-ASCII letters are allowed, but for code portability it is better to stick to “a” to “z” (and “A” to “Z”). The help page `?make.names` gives precise details about what is and isn’t allowed.

The spaces around the assignment operators aren’t compulsory, but they help readability, especially with `<-`, so we can easily distinguish assignment from less than:

```
x <- 3
x < -3
x<-3     #is this assignment or less than?
```

We can also do global assignment using `<<-`. There’ll be more on what this means when we cover environments and scoping in Environments in Chapter 6; for now, just think of it as creating a variab

- click Å’uvres complÃ¨tes, tome 6 : La Somme athÃ©ologique II: Sur Nietzsche - MÃ©morandum - Annexes
- *read online Authentic Mexican: Regional Cooking from the Heart of Mexico (20th Anniversary Edition)*
- Getting Started with Arduino (1st Edition) pdf
- **Mercy pdf**
- *click Sagesses d'hier et d'aujourd'hui*
- *read online Educating Nurses: A Call for Radical Transformation for free*

- http://bestarthritiscare.com/library/--uvres-compl--tes--tome-6---La-Somme-ath--ologique-II--Sur-Nietzsche---M--morandum---Annexes.pdf
- http://cambridgebrass.com/?freebooks/The-Art-of-Happiness--A-Handbook-for-Living--10th-Anniversary-Edition-.pdf
- http://damianfoster.com/books/Getting-Started-with-Arduino--1st-Edition-.pdf
- http://conexdxb.com/library/Mercy.pdf
- http://growingsomeroots.com/ebooks/Some-of-My-Lives--A-Scrapbook-Memoir.pdf
- http://anvilpr.com/library/Educating-Nurses--A-Call-for-Radical-Transformation.pdf