Beginning Computer Science with R

Homer White

2017-10-21

Contents

Preface

Welcome to *Beginning Computer Science with R*! These notes will be your primary source for CSC 115: Computer Science I, offered at Georgetown College. They will also carry you part of the way through CSC 215. A supplementary text, *Hands-On Programming with R* by Garrett Grolemund (Grolemund, 2014), will be used in CSC 115 so you should make sure that you have a copy.

For the first semester you won't need a computer of your own: you can do all of your work on the R Studio Server, which you will access with your College network username and password. Eventually, though, you will need to learn how to install and maintain professional software development tools on your own machine, so at some point early on in CSC 215 you will install R, R Studio and various other tools on a laptop of your own. Then you will bring the laptop to class for your daily work.

These Notes are available not only on the web but also as a PDF document that can be downloaded from the website (click the PDF icon, found among the icons at the top of the page). The PDF is useful when you are working offline, and it also has an index.

Instructional videos on selected topics will be published from time to time on this YouTube Channel: `https://www.youtube.com/user/GCstats`, in the CSC playlist.

These notes are about the R programming language as such, so although in the class we will work within the R Studio Integrated Deveopment Environment right from the start, R Studio is not directly covered here. Eventually we will begin to write documents in R Markdown, which is also not treated in these Notes. For a resource on these topics in written form that will supplement class instructon and the videos on our YouTube Channel, you might want to consult the excellent little book *Getting used to R, RStudio, and R Markdown*[1] (Ismay, 2016).

0.1 The Why of these Notes: Remarks for Colleagues

There is a plethora of books on R, covering pretty much every domain of application of the language, from ecology to spatial statistics to machine learning and data science. There are even some books—among the very finest of R-books, in my view—on R as a a programming language. Why then, write yet another book?

It all has to do with my particular situation. As Fate and the vagueries of small-college staffing would have it, I find myself preparing to teach the minor in Computer Science at the liberal arts College where for many years I had previously taught mathematics and statistics. As I considered how to redesign a minor lean enough to be staffed by one not-quite-full-time person (I still handle some mathematics) and at the same offer something of absorbing interest to a reasonable number of undergraduates, I realized that it could not simply be a truncated version of a major: a year of programming in a strongly-typed, compiled, large-scale programming language such as C++ or Java, followed by algorithms, data structures, compilers and perhaps a bit of theory. For one thing, I shy away from large-scale languages: I tend to want my results fast and I want to build something interesting as soon as possible, and for the most part beginning students feel the same way. About the only thing a minor as a curtailed major would prepare them for is, well,

[1] `https://bookdown.org/chesterismay/rbasics/`

completion of the major—which unfortunately we do not have. For another thing, my background simply isn't in Computer Science. I am by training a mathematician with a specialty in ergodic theory, which could be considered an extreme generalization of probability theory. Computer programming, with its fetish for the careful organization of information and its stress on practices to manage complexity, simply isn't in my theoretical blood.

On the other hand in my years of teaching and practicing statistics I have gotten to be a half-good programmer by recognizing, through hard experience, the value of learning to program well from the ground up. I have become increasingly fascinated by the computational aspects of statistics and by the entire computer-based workflow of data analysis—from data collection and data munging to analysis and reporting—and by the paradigm of reproducible research that can be incorporated into nearly every stage of that workflow. With the tendency of data analysis to move toward the web I'm also seeing the utility of being aware of web design issues as a data analyst, and, as a web designer and developer, of being cognizant of trends in internet-based data collection and data reporting.

Accordingly the minor will have a dual focus: data analysis and web design, with a bit of back-end web development thrown in to unify the two domains. Students will undertake a serious study of two major scripting languages: R for data analysis and JavaScript for web programming, both from a fairly systematic programming point of view, with due attention to procedural, object-oriented and (to a lesser extent) functional programming paradigms.

The question is: which language to use in the freshman year? Some institutions are moving toward JavaScript: in fact Stanford University will pilot JavaScript in several sections of its introductory CS course in the Fall of 2017. There are certainly considerations in favor of a JavaScript-first approach: it's a popular language with Node available as an interactive run-time environment and the browser as a environment in which exciting applications can be built quickly. And whereas R is less widely-used and is still considered a domain-specific language, JavaScript can rightly be said to have made the leap into the ranks of general-purpose programming languages. R also has the reputation of being a prickly language with a somewhat inconsistent syntax and with documentation that is "expert-friendly" at best.

On the other hand R is designed for one-line interactivity at the console, so it's possible for a beginner to get simple programs working quickly. The R-ecosystem has also become a lot more user-friendly in recent years. The RStudio[2] IDE is comparable to top-flight integrated development environments for many other major languages and yet is still relatively lightweight and accessible to beginners. The Server version of R Studio is especially useful for new programmers, as it saves them from having to deal with installation and other IT issues on their own machines, permitting them to focus on coding. It's also easy, in a server setting, to make class materials available and to collect and return assignments. R Markdown is fine platform for producing course notes (this book is written in R Markdown with the excellent **bookdown** package (Xie, 2017b)) and slides as well. Students, too, can use R Markdown to both write and discuss their programs in a single document. The **blogdown** package (Xie, 2017a) permits students to begin writing for the public about technical programming issues—or about anything at all, really, as more than a few of them are taking majors in the Humanities—thus building up an professional resume of online work. Finally, the **shiny** package (Chang et al., 2017) permits students to build simple interactive web apps for data analysis that can be used by non-coders. Both **blogdown** and **shiny** prompt students to consider early on—even in the first year, if the pacing is right—concepts of web design, the other primary focus of the minor.

Hence the choice was made to teach a first-year computer science course, to beginning programmers, with R. As I pointed out earlier, there do exist some excellent books on R as a programming language that do not presume previous experience with R. One example is Norman Matloff's *The Art of R Programming* (Matloff, 2011). Matloff, however, presumes that the reader either has prior programming experience in some other language or else possesses sufficient computational maturity, acquired perhaps through extensive prior training in the mathematical sciences. Another great text is Garrett Grolemund's *Hands-on Programming with R* (Grolemund, 2014). Grolemund's book is lively and to-the-point and starts off with excellent motivating examples. Grolemund is also a master explainer, and he has put considerable effort into visual representation of programming concepts such as element-wise operations on vectors and the enclosure-relationships between

[2] `https://www.rstudio.com/products/rstudio/`

environments; for that reason his text is being used as a supplementary textbook for this course. On the other hand, even though he doesn't assume that the reader has prior coding experience, Grolemund does assume some prior background in data analysis and a strong motivation, on the reader's part, to persevere with nontrivial R-programming issues such as lexical scoping in the hopes of eventual payoffs in programming for data science. In short, Grolemund also assumes more computational maturity than will be usually be found among beginning programmers at many small liberal arts colleges.

Hence the niche for the notes (or perhaps text-under-development) offered here. I aim to be a bit more copious and slow-paced than Grolemund and a bit less sophisticated than Matloff. These notes will also contain a more extensive set of problems, ranging in difficulty from practice exercises to fairly extended projects that students might write up in R Markdown documents.

Experienced programmers and R enthusiasts will be struck by the absence of certain topics. Programmers will observe that there is no real attention to algorithms (sorting is just `sort()` or `order()`), and although functions receive lots of attention there is no mention of recursion. In future editions I might cover recursion, as I believe that it is wonderful for the development of thinking skills, but it's not likely that a web developer or data analyst would have the need to write a recursive function. Time spent on recursion and on various efficient algorithms for sorting and searching may be better spent, in my view, on extended programming projects, Shiny apps and blogging, and introductions to tools of the programmer's trade such as version control and GitHub. I hope by the end of the first year to have made time for all of these "extra" topics. R-specific tools such as **dplyr** for data manipulation, the various packages for web scraping and database interface, as well as packages for machine learning, are considered in later courses in the minor that are more specifically concerned with data analysis.

Two of the most fundamental topics in any comprehensive discussion of the R language—lexical scoping and computing on the language—are absent from this book. Lexical scoping and its implications are mentioned only in a brief footnote. Partly this is due the fact that most of the elementary applications of lexical scoping mentioned in the literature are related to scientific computing, which won't be a concern for most of my students. Certainly lexical scoping is important for understanding how R-packages work, but elementary students don't author packages. As for computing on the language it is true that users are affected by it all the time (e.g., whenever they use functions with a formula interface), but generally one need not perform any computation on the language until one begins writing formula-interface functions for the benefit of casual R-users.

On the other hand I have made some effort to explore programming paradigms other than procedural programming, perhaps in a bit more depth than in other elementary texts that teach with a scripting language. There is a chapter on functional programming that, although it admittedly does not get far into the functional paradigm, at least does treat extensively of R's support for higher-order functions. A chapter on object-oriented programming covers not only the generic-function OO that has been with R from the start but also an implementation of message-passing OO (Winston Chang's **R6** package (Chang, 2017)). My hope is that these topics will not only sharpen my students' R-programming skills but also prepare them for encounters with the OO-methods and higher-order functions that are ubiquitous in JavaScript. Finally, there is a pretty serious chapter on regular expressions, because:

- they are useful in data analysis;
- I have not found a treatment of regular expressions in R that a person without significant prior exposure to them in other languages has a prayer of following;
- and because if you master regular expressions then you feel like a wizard.

As for the numerous Wizard of Oz-themed examples, I can offer no defense other than haste. Perhaps a future version will be built around something more suitable—the Harry Potter series, say.

0.2 History of R

The story of R begins at Bell Labs[3] in 1975, with the development, by John Chambers and several other colleagues, of the S language for statistical computing. The language became well-known among statisticians and data analysts, especially in the academic community.

In the early 1990's Ross Ihaka of the University of Auckland in New Zealand was making a study of the Scheme language as described in the classic MIT textbook *Structure and Interpretation of Computer Programs* (Harold Abelson and Sussman, 1996), and was impressed with the possibilities of the language for data analysis applications. Desiring to build a free analysis tool for his graduate students, Ihaka teamed up with his Auckland colleague Robert Gentleman and the two gradually developed a language with an external syntax similar to S but with an underlying engine based heavily upon Scheme. Because of the similarity with the better-known S—or, by some account, because of the initial letter in the first names of both men—they named their new language "R".

Initially the two men believed that their work on the new language was scarcely more than "playing games" and that it would not be used outside of the University of Auckland. Eventually, though, the two placed a small announcement of their project on the email list *s-news* and began to draw the interest of other statisticians, including Martin Machler of the Swiss Federal Institute of Technology in Zurich, Switzerland. Machler saw great potential for R, and in 1995 persuaded Ihaka and Gentleman to release it as "free software" under a GNU Public License[4]. The decision to make R free stimulated further interest in the language and encouraged many experts in statistical computation to become involved in its further development.

The first official public release of R (version 1.0.0) occurred on February 29, 2000. Since that time R has grown in popularity at an increasing rate, to the point where it is by now one of the world's most widely-used domain-specific computer languages, ranking among the top dozen computer languages overall.

Many people have contributed to the development of R. As of the composition of this History, the Comprehensive R Archive Network (CRAN) now hosts 10,633 contributed packages, each of which aims to extend the capabilities of R in a specific way. R is usually the platform in which new statistical techniques are first implemented by the researchers who develop them. It is widely used in the sciences, business and finance.

0.3 Acknowledgements

I am greatly indebted to:

Norman Matloff and *Hadley Wickham* for their excellent foundational books ((Matloff, 2011), (Wickham, 2014)) on the R language. *Garrett Grolemund*, for his informal but precise expository style, as exemplified in numerous R Studio webinars and in (Grolemund, 2014).

Allen Downey, for *Think Python* (Downey, 2015). This book formed my ideas about ordering and selection of topics for computer science at an elementary level, and helped me think about teaching computer science in a way that was as independent as possible from the specific language of instruction.

Everyone at R Studio, including Hadley, *Yihui Xie* for R Markdown and the family of R Markdown-related packages, *Joe Cheng* for conceiving and pioneering **shiny**, *Winston Chang* for **shiny** and **R6**, and of course *JJ Allaire* for developing the IDE and calling together the remarkable constellation of developers and evangelists who have contributed so much to the R community and ecosystem.

Danny Kaplan, *Nick Horton* and *Randall Pruim* for pioneering the Mosaic Project that has enabled so many faculty to teach undergraduate statistics with R. I am especially indebted to Danny for curricular inspiration and to Randall for R-programming advice in the incipient stages of my journey as an R-developer. Nick has been a great encourager of everyone associated with the Mosaic community.

[3] `https://en.wikipedia.org/wiki/Bell_Labs`

[4] `https://www.gnu.org/licenses/gpl-faq.html#WhatDoesGPLStandFor`

My former colleague *Rebekah Robinson* who, upon learning of Mosaic and R Markdown, insisted that at Georgetown we should figure out how to teach elementary statistics with R.

My colleagues *William Harris* and *Christine Leverenz*, who learned to teach in the R way.

My students, especially *Woody Burchett*, *Luke Garnett*, *Shawn Marcom*, *Jacob Townsend* and *Andrew Giles*, for work with me on various R-related research projects. Luke has gone on to become a valued colleague at Georgetown.

Georgetown College, for granting me the sabbatical time in Spring 2017 to work on these Notes, and on other programming topics prerequisite to teaching Computer Science.

My wife *Mary Lou* and daughters *Clare*, *Catherine* and *Agnes*, for patience and support. By now they have heard quite enough about programming.

0.4 Miscellaneous Information

0.4.1 Typographical Conventions

Computer code, whether within a line of text or as displayed text, appears like this: `code snippet`.

Identifiers are represented as is (e.g., `variableName`, if,else,while, `etc.) except for the names of functions, whihc are followed by a pair of parentheses in order to stress their status as functions. Thus we write`length()`for the`length‘-function.

Displayed text representing output to the console appears with double hash-tags at the beginning of each line, thus:

```
## 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.
```

The hashtags themsevles are not present in the output itself.

Names of R-package are in boldface, thus: package **devtools**.

Terms are italicized when they are first introduced, e.g.: “R follows a set of internal rules to *coerce* some of the values to a new type in such a way that all resulting values are of the same type.” Italicization can also indicate emphasis.

0.4.2 License

Chapter 1

Our First Computer Programs

Figure 1.1: The Yellow Brick Road, by Jorge Cruz. clipartfest.com.

1.1 Hello, World!

Let's write our very first computer program. Type the following code into the console and press **Enter**:

```
cat("Hello, World!")
```

In the console you should see the following output:

```
## Hello, World!
```

That's it—you just wrote a *computer program*. A computer program is simply a sequence of instructions tha perform a specific task when they are *executed* (carried out) by the computer. In the above example, there was only one instruction in the sequence: it was the command to reproduce the *string* "Hello, World" in the console exactly as it is. The name of that command is `cat()`.

Let's try another small program.

Type the following code into the console and press **Enter**:

```
cat(2+2)
```

In the console you should see the following output:

```
## 4
```

We have been using the `cat()` function so far. Actually you can can get output without it. For example, suppose you type:

```
"Hello, World!"
```

Then in the console you see:

```
## [1] "Hello World"
```

There are quote-marks around the text: that's not so pretty, but we still get the basic output.

Similarly, you can try:

```
2+2
```

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

Notice that when we don't use `cat()` the output begins with a strange `[1]`. In the next chapter we'll learn why this happens; for now, just ignore it.

1.2 A Quick Tour

We now embark on a tour of some of R's basic capabilities. In later chapters we will examine in detail the programming concepts that underlie the features we explore now.

1.2.1 Basic Arithmetic

R can be treated like a calculator. You can:

- add numbers (+)

- subtract numbers (-)
- multiply numbers (*)
- divide numbers (/)
- raise a number to a power (^)

Just as on a graphing calculator, parentheses can be used to clarify the order of operations.

Here are some examples:

To get $\frac{27-10}{3}$, use:

```
(27-3)/10
```

```
## [1] 2.4
```

To get $3^2 + 4^2$ try:

```
3^2+4^2
```

```
## [1] 25
```

Sometimes you'll want to take roots. As with a calculator, you can accomplish this by raising your number to a fractional power. So if you want $\sqrt[3]{64}$ then you could try:

```
64^(1/3)
```

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

If you would like square roots then you can either raise your number to the 1/2-power or you could use R's special square-root function:

```
sqrt(64)
```

```
## [1] 8
```

One way or another, you can evaluate quite complex mathematical expressions. For example, to get $\sqrt{3^2 + 4^2}$ simply type:

```
sqrt(3^2+4^2)
```

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

1.2.2 Read-Evaluate-Print-Loop

So far you have been using R in what computer scientists call *interactive mode*. This means that you type something in at the console; R immedidately reads what you type and evaluates it, and prints the resulting value to the console for you to see. Then you type something else, and so on. This back-and-forth process is often called the *Read-Evaluate-Print-Loop*, or REPL for short. R is one of several computer languages that make it easy for you to see the results of its computations in the console. That's because it was originally designed for use by statisticians and data analysts, who often want to run a small procedure, check on the results and then try a new or related procedure and check on the results … until their analysis is complete. From our point of view as beginning programmers, though, the REPL makes it easy to see what R is doing and to get immediate feedback on the very simple programs that we are now writing.

1.2.3 Variables

Quite often you will want to use the same value several different times. You can so this by creating a *variable* with the *assignment operator* `<-`.

```
a <- 10
```

The previous statement puts the value 10 in the computer's memory and causes the name `a` to be *bound* to it. This means that if you ask R to show you `a`, you'll get that value:

```
a
```

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

Now you can use `a` as much as you like. Whenever you use it, R will know that it stands for the value 10:

```
a + 23
```

```
## [1] 33
```

```
sqrt(a)
```

```
## [1] 3.162278
```

Later on if you want to bind the name `a` to a different value, you can do so, with another assignment-statement:

```
a <- 4
a + 23
```

```
## [1] 27
```

Let's write some code to introduce creatures of various types. A creature should give his or her name, say what type of creature he or she is, and name a favorite food.

```
creatureType <-"Munchkin"
creatureName <- "Boq"
creatureFood <- "corn"
```

Notice that I chose variable-names that are descriptive of the values to which they are bound. That's often a good practice.

Next, let's combine our items into a greeting:

```
paste("Hello, I am a ",
      creatureType,
      ". My name is ",
      creatureName,
      ".  I like to eat ",
      creatureFood,
      ".",
      sep = "")
```

```
## [1] "Hello, I am a Munchkin. My name is Boq.  I like to eat corn."
```

We see that `paste()` function puts strings together. The `sep = ""` argument at the end specifies that no space is to be inserted between the strings when they combined.

Another thing we notice in the previous code is that R can ignore white space: we were able to place the parts of the command on different lines. This helps prevent our lines from being too long, and allows us to arrange the code so that it's easy to read.

Spaces do matter inside a string, though:

```
kalidah <- "Teddy"
kalidah
```

```
## [1] "Teddy"
```

```
kalidah2 <- "Ted  dy"
kalidah2
```

```
## [1] "Ted  dy"
```

You must also be careful not to insert spaces within the name of any object:

```
kali dah2
```

```
## Error: unexpected symbol in "kali dah2"
```

R got confused by the unexpected space: it knows about the name `kalidah2`, but `kali dah2` means nothing to R.

Getting back to the Oz-creatures: it would be nice if a creatures's greeting could be split over several lines. This is possible if you use the special string "\n", which produces a newline. Just incorporate it into your message, as follows:

```
paste("Hello, I am a ",
      creatureType,
      ".\nMy name is ",
      creatureName,
      ".\nI like to eat ",
      creatureFood,
      ".",
      sep = "")
```

```
## [1] "Hello, I am a Munchkin.\nMy name is Boq.\nI like to eat corn."
```

That doesn't look like an improvement at all! But what if we were to `cat()` it?

```
message <- paste("Hello, I am a ",
                 creatureType,
                 ".\nMy name is ",
                 creatureName,
                 ".\nI like to eat ",
                 creatureFood,
```

```
                  ".",
                  sep = "")
cat(message)
```

```
## Hello, I am a Munchkin.
## My name is Boq.
## I like to eat corn.
```

That's much nicer.

That last example showed that you can use variables together with functions to create new variables. Here is another example:

```
a <- 10
b <- 27
mySum <- a + b
mySum
```

```
## [1] 37
```

1.2.4 Functions

Let's say that we want to introduce George the Quadling. We might try:

```
creatureName <- "George"
creatureType <- "Quadling"
creatureFood <- "cookies"
cat(message)
```

```
## Hello, I am a Munchkin.
## My name is Boq.
## I like to eat corn.
```

Hmm, that didn't go so well: we got Boq instead. The problem is that the variable`message` was created using the original values of `creatureName`, `creatureType` and `creatureFood`, not the new values that we are interested in. To do it right we should have re-made `message`, as follows:

```
creatureName <- "George"
creatureType <- "Quadling"
creatureFood <- "cookies"
message <- paste("Hello, I am a ",
                  creatureType,
                  ".  \nMy name is ",
                  creatureName,
                  ".\nI like to eat ",
                  creatureFood,
                  ".",
                  sep = "")
cat(message)
```

```
## Hello, I am a Quadling.
## My name is George.
```

```
## I like to eat cookies.
```

That's great, but it seems that every time we introduce a new creature we have to type a lot of code. It would be much better if we could find a way to re-use code, rather than repeating it.

Functions allow us to re-use code. Let's define a function to do introductions:

```
intro <- function(name, type, food) {
  message <- paste("Hello, I am a ",
                   type,
                   ".  \nMy name is ",
                   name,
                   ".\nI like to eat ",
                   food,
                   ".",
                   sep = '')
  cat(message)
}
```

In the console nothing happens. We only created the function `intro()`, we haven't *called* it yet. Let's call `intro`:

```
intro(name = "Frederick", type = "Winkie", food = "macaroni")
```

```
## Hello, I am a Winkie.
## My name is Frederick.
## I like to eat macaroni.
```

R allows you to be lazy: you can omit the *parameters* `name`, `type` and `food`, so long as you indicate what their values should be, in the correct order:

```
intro("Frederick", "Winkie", "macaroni")
```

```
## Hello, I am a Winkie.
## My name is Frederick.
## I like to eat macaroni.
```

1.2.5 Data and Graphics

Anyone can use R, but it was created for statisticians, so it has many features that are helpful in data analysis. Let's take a quick look at a data set from a *contributed R package*, the package **mosaicData** (Pruim et al., 2016).

First, we'll *attach* the package, so R can find all of the goodies it contains:

```
library(mosaicData)
```

Package **mosaicData** contains a number of interesting datasets that are useful in the teaching of statistics. Let's look into one of them—`Births78`—using R's `help()` function:

```
help("Births78")
```

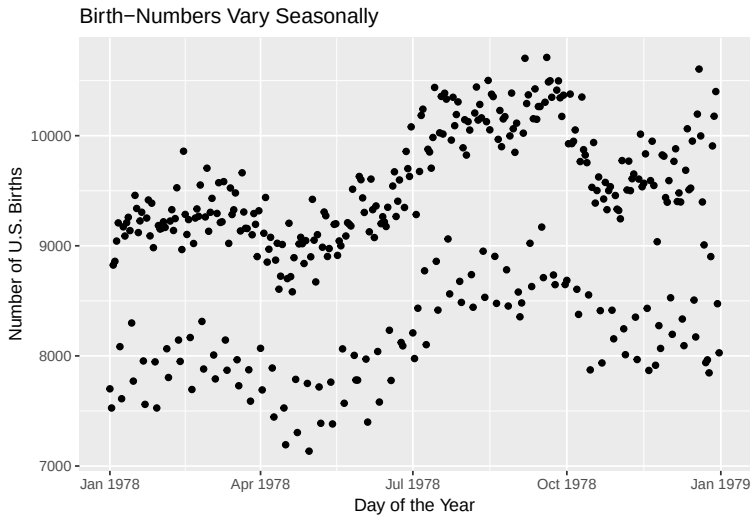


Figure 1.2: A simple scatterplot with R's ggplot2 graphics system.

We learn that `Births78` is a *data frame* containing information on the number of births each day, during the year 1978. (A data frame is one of R's most important *data structures*. We'll learn more about them in Chapter 7.) The frame has 365 rows, one for each day in the year, and four columns. Each column contains the values of a variable recorded for each day:

- the calendar `date` of that day;
- `births`: the number of children born in the United States on that day;
- `dayofyear`: the number of the day within the year 1978 (1 being January 1, 2 being January 2, and so on);
- `wday`: the day of week for that day (Sunday, Monday, etc.).

We can view the first few row of the data frame using R's `head()` function:

```
head(Births78, n = 10)
```

```
##          date births dayofyear  wday weekend
## 1  1978-01-01   7701         1   Sun weekend
## 2  1978-01-02   7527         2   Mon weekday
## 3  1978-01-03   8825         3  Tues weekday
## 4  1978-01-04   8859         4   Wed weekday
## 5  1978-01-05   9043         5 Thurs weekday
## 6  1978-01-06   9208         6   Fri weekday
## 7  1978-01-07   8084         7   Sat weekend
## 8  1978-01-08   7611         8   Sun weekend
## 9  1978-01-09   9172         9   Mon weekday
## 10 1978-01-10   9089        10  Tues weekday
```

We might wonder whether the number of births varies with the time of year. One way to investigate this question is to make a *scatterplot*, where the days of the year (numbered 1 through 365) are on the horizontal axis and the number of births for each day are on the vertical axis. Figure 1.2 shows such a plot.[1]

Clearly the number of births varies seasonally: more babies are born in late summer and early fall, whereas spring births are not as frequent. But there is something mysterious about the plot: Why do there are appear to be two clearly separated groups of days, one with considerably more births than the other? What

[1]The plot is made with the **ggplot2** graphics package (Wickham and Chang, 2017). Graphing will not be a major focus of the course at first, but we will return from time to time, to the subject of graphing in **ggplot2** as our need for graphs dictates.

is going on here? As we learn to program in R, we will gradually acquire the skills needed to answer this and many other intriguing questions.

1.3 Debugging

It's easy to make mistakes when you program—even when you are very experienced! Incorrect computer code is said to have a *bug*, and the art of finding bugs and correcting them is called *debugging*.

Consider the following code:

```
scarecrowQuote <- "It is such an uncomfortable feeling to know one is a fool."
paste("The Scarecrow says: ", scarecrowquote)
```

When we run it we get the following error message in the console:

```
## Error in paste("The Scarecrow says: ", scarecrowquote) :
##  object 'scarecrowquote' not found
```

R's error messages are often quite mysterious—they are intended to be maximally useful to experienced R programmers—but it's always a good idea to read them anyway. In this case the message clearly tells us the problem: R cannot find the object `scarecrowquote` on its search path. This prompt us to look more closely at the name `scarecrow`, and sooner or later we will realize that we have a misspelling: the variable that was actually defined was `scarecrowQuote`, with a capital Q.

The correct code is:

```
scarecrowQuote <- "It is such an uncomfortable feeling to know one is a fool."
paste("The Scarecrow says: ", scarecrowQuote)
```

```
## [1] "The Scarecrow says:  It is such an uncomfortable feeling to know one is a fool."
```

Always bear in mind that R is case-sensitive!

Here's another buggy bit of code:

```
SermonMountComment <- paste("Oh, it's "blessed are the meek."",
                      "\nI'm glad they are getting something:\n",
                      "they have a hell of a time.")
cat(SermonMountComment)
```

The idea is to produce:

```
## Oh, it's "blessed are the meek.
## I'm glad they are getting something:
##  they have a hell of a time.
```

But when we run the code we get the following result instead:

```
> rm(SermonMountComment)
> SermonMountComment <- paste("Oh, it's "blessed are the meek."",
Error: unexpected symbol in "SermonMountComment <- paste("Oh, it's "blessed"
>                             "\nI'm glad they are getting something: ",
Error: unexpected ',' in "                        "\nI'm glad they are getting something: ","
>                             "they have a hell of a time.")
```

```
Error: unexpected ')' in "                              "they have a hell of a time.")"
> cat(SermonMountComment)
Error in cat(SermonMountComment) : object 'SermonMountComment' not found
```

This can be a bit more difficult to read. The problems appear to start near the beginning of the construction of the string `SermonMountComment`.

After looking at it a while we focus on the first string argument to the `paste()` function:

```
"Oh, it's "blessed are the meek.""
```

We see that this string has quotes within quotes. Now R uses quotes as *delimiters* for strings: that is, quote-marks indicate where a string begins and where it ends. Hence from R's point of view, the first string consists of just: `"Oh, it's "`. But then there is no comma to separate this string from the next string argument that the `paste()` functions expects. Instead R sees the `b` in `blessed`; that's an *unexpected symbol*. Things go downhill from there.

There are a couple of ways to correct the problem. One approach is to use single quotes inside any string that is delimited with double quotes, thus:

```
SermonMountComment <- paste("Oh, it's 'blessed are the meek.'",
                      "\nI'm glad they are getting something:\n",
                      "they have a hell of a time.")
cat(SermonMountComment)
```

```
## Oh, it's 'blessed are the meek.'
## I'm glad they are getting something:
##  they have a hell of a time.
```

On the other hand if you really want those double-quotes inside the string, you can *escape* their special meaning as string-delimiter by prepending a backslash (\) to them, thus:

```
SermonMountComment <- paste("Oh, it's \"blessed are the meek.\"",
                      "\nI'm glad they are getting something:\n",
                      "they have a hell of a time.")
cat(SermonMountComment)
```

```
## Oh, it's "blessed are the meek."
## I'm glad they are getting something:
##  they have a hell of a time.
```

There are a number of special characters that are formed by "escaping" the usual meaning of some other character. Some common examples are:

- `\n`: produces a newline instead of `n`
- `\t`: produces a tab-space instead of `t`
- `\"`: produces an actual quote-mark, instead of beginning or ending a string.

Strings are a tricky topic in any computer programming language: in fact we will devote all of Chapter 10 to them.

Glossary

Computer Program A sequence of instructions that performs a specific task when executed by a computer.

String A value in a computer program that constitutes text (as opposed to numbers of some other type of data).

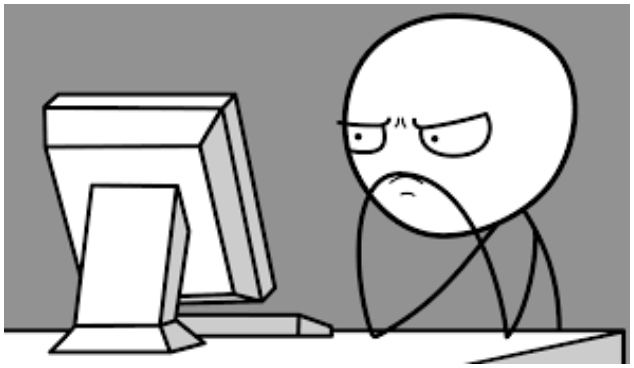
Interactive Mode A type of engagement between a human and a computer in which the computer prompts the humand for data and/or commands and may respond with output that the human can read and/or interpret.

Read-Evaluate-Print Loop An interactive cycle in which the R-interpreter reads an expression from the console, evaluates it, and prints out the value to the console.

Data Structure A particular way of organizing information in an computer program so that it can be used efficiently.

Delimiter A character in a programing languages that is used to mark the beginning and/or end of a value.

Exercises



1. Write a program that modifies the function `intro()` (see Section 1.2.4) so that the person who introduces him or herself states a favorite sport. For example, the result of the following function call:

```
intro(name = "Bettina", type = "human", sport = "lacrosse")
```

   should be:

```
## Hello, I am a human.
## My name is Bettina.
## My favorite sport is lacrosse.
```

2. Write a program to produce the following output to the console:

```
## *
## **
## ***
## **
## *
```

3. Suppose we want to `cat` "Hello, World" to the console, and we enter:

```
cat(Hello, World!)
```

   What does R say? What did we do wrong?

Chapter 2

Vectors

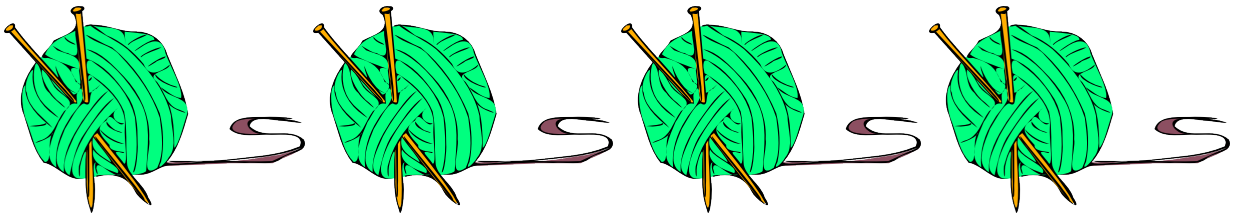


Figure 2.1: 'rep(yarn, times = 4)'.

This Chapter gets you started officially with R. While the theme is *vectors*, the most important data structure in R, we'll learn also about variables and variable names, vector types, reserved words, assignment and many of R's basic operators.

2.1 What is a Vector?

If you have heard of vectors before in mathematics, you might think of a vector as something that has a magnitude and a direction, and that can be represented by a sequence of numbers. In its notion of a vector, R keeps the idea of a sequence but discards magnitude and direction. The notion of "numbers" isn't even necessary.

For R, a vector is simply a sequence of elements. There are two general sort of vectors:

- *atomic* vectors that come in one of six forms called *vector types*;
- non-atomic vectors, called *lists*, whose elements can be any sort of R-object at all.

For now we'll just study atomic vectors. Let's make a few vectors, as examples.

We can make a vector of numbers using the `c()` function:

```
numVec <- c(23.2, 45, 631, -273, 0, 48.371, 100000,
            85, 92, -236, 8546, 98774, 0, 0, 1, 3)
numVec
```

```
##  [1]     23.200     45.000    631.000   -273.000      0.000     48.371
##  [7] 100000.000     85.000     92.000   -236.000   8546.000  98774.000
## [13]      0.000      0.000      1.000      3.000
```

You can think of `c` as standing for "combine." `c()` takes its arguments, all of which are separated by commas, and combines them to make a vector.

If you closely examine the above output, you'll notice that R printed out all of the numerical values in the vector to three decimal places, which happened to be the largest number of decimal places we assigned to any of the numbers that made up `numVec`. You'll also notice the numbers in brackets at the beginning of the lines. Each number represents the position within the vector occupied by the first element of the vector that is printed on the line. The position of an element in a vector is called its *index*. Reporting the indices of leading elements helps you locate particular elements in the output.

2.1.1 Types of Atomic Vectors

The numbers in `numVec` are what programmers call *double-precision* numbers. You can verify this for yourself with the `typeof()` function:

```
typeof(numVec)
```

```
## [1] "double"
```

The `typeof()` function returns the type of any object in R. As far as vectors are concerned, there are six possible types, of which we will deal with only four:

- `double`
- `integer`
- `character`
- `logical`

Let's look at examples of the other types. Here is a vector of type `integer`:

```
intVec <- c(3L, 17L, -22L, 45L)
intVec
```

```
## [1]   3  17 -22  45
```

The `L` after each number signifies to R that the number should be stored in memory as an integer, rather than in double-precision format. Officially, the type is `integer`:

```
typeof(intVec)
```

```
## [1] "integer"
```

You should know that if you left off one or more of the `L`'s, then R would create a vector of type `double`:

```
numVec2 <- c(3, 17, -22, 45)
typeof(numVec2)
```

```
## [1] "double"
```

We won't work much with integer-type vectors, but you'll see them out in the wild.

We can also make vectors out of pieces of text called *strings*: these are called *character* vectors. As noted in the previous chapter, we use quotes to delimit strings:

```
strVec <- c("Brains", "are", "not", "the", "best",
            "things", "in", "the", "world", "93.2")
strVec
```

```
##  [1] "Brains" "are"    "not"    "the"    "best"   "things" "in"
##  [8] "the"    "world"  "93.2"
```

```
typeof(strVec)
```

```
## [1] "character"
```

Notice that `"93.2"` makes a string, not a number.

The last type of vectors to consider are the `logical` vectors. Here is an example:

```
logVec <- c(TRUE, FALSE, T, T, F, F, FALSE)
logVec
```

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

In order to represent a logical value you can use:

- `TRUE` or `T` to represent truth;
- `FALSE` or `F` to represent falsity.

You can't represent truth or falsity any other way. If you try anything else—like the following—you get an error:

```
badVec <- c(TRUE, false)
```

```
## Error: object 'false' not found
```

Note: Although R allows `T` to be interpreted as `TRUE` and `F` as `FALSE`, it can be dangerous to use them in some circumstances. Best to get into the habit of always using `TRUE` and `FALSE`, rather than the permitted abbreviations.

2.1.2 Coercion

What would happen if you tried to represent falsity with the string `"false"`?

```
newVector <- c(TRUE, "false")
newVector
```

```
## [1] "TRUE"  "false"
```

`newVector` is not a logical vector. Check it out:

```
typeof(newVector)
```

```
## [1] "character"
```

In order to understand what just happened here, you must recall that all of the elements of an atomic vector have to be of the same type. If the `c()` function is presented with values of different types, then R follows a set of internal rules to *coerce* some of the values to a new type in such a way that all resulting values are of the same type. You don't need to know all of the coercion rules, but it's worth noting that

- `character` beats `double`,
- which in turn beats `integer`,
- which in in turn beats `logical`.

The following examples show this:

```
typeof(c("one", 1, 1L, TRUE))
```

```
## [1] "character"
```

```
typeof(c(1, 1L, TRUE))
```

```
## [1] "double"
```

```
typeof(c(1L, TRUE))
```

```
## [1] "integer"
```

Automatic coercion can be convenient in some circumstances, but in others it can give unexpected results. It's best to keep track of what types you are dealing with and to exercise caution when combining values to make new vectors.

You can also coerce vectors "manually" with the functions:

- `as.numeric()` ;
- `as.integer()` ;
- `as.character()` ;
- `as.logical()` .

Here are some examples:

```
numVec <- c(3, 2.5, -7.32, 0)
as.character(numVec)
```

```
## [1] "3"     "2.5"   "-7.32" "0"
```

```
as.integer(numVec)
```

```
## [1]  3  2 -7  0
```

```
as.logical(numVec)
```

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

Note that in coercion from numerical to logical, the number 0 becomes `FALSE` and all non-zero numbers become `TRUE`.

2.1.3 Combining Vectors

You can combine vectors you have already created to make new, bigger ones:

```
numVec1 <- c(5, 3, 10)
numVec2 <- c(1, 2, 3, 4, 5, 6)
numCombined <- c(numVec1, numVec2)
numCombined
```

```
## [1]  5  3 10  1  2  3  4  5  6
```

You can see here that vectors are different from sets: they are allowed to repeat the same value in different indices, as we see in the case of the 3's above.

2.1.4 NA Values

Consider the following vector, which we may think of as recording the heights of people, in inches:

```
heights <- c(72, 70, 69, 58, NA, 45)
```

The `NA` in the fifth position of the vector is a special value that may be considered to mean "Not Assigned." It's R's way of letting us indicate that a value was not recorded or has gone missing for some reason.

2.1.5 "Everything in R is a Vector"

Some folks say that everything in R is a vector. That's a bit of an exaggeration but it's remarkably close to the truth.

And yet it seems implausible. What about the elements of an atomic vector, for instance? A single element doesn't look at all like a vector: it's a value, not a *sequence* of values.

Or so we might think. But really, in R there are no "single values" that can exist by themselves. Consider, for instance, what we think of as the number 17:

```
17
```

```
## [1] 17
```

See the `[1]` in front, in the output above? It indicates that the line begins with the *first element* of a vector. So 17 doesn't exist on its own: it exists a vector of type `double`—a vector of length 1.

Even `NA` is, all along, a vector of length 1

```
NA
```

```
## [1] NA
```

It is of type `logical`:

```
typeof(NA)
```

```
## [1] "logical"
```

Note that even the type of `NA` evaluates, in R, to a vector: a character vector of length 1 whose only element is the string "logical"!

2.1.6 Named Vectors

The elements of a vector can have names, if we like:

```
ages <- c(Bettina = 32, Chris = 64, Ramesh = 101)
ages
```

```
## Bettina   Chris  Ramesh
##      32      64     101
```

Having names doesn't keep the vector from being a vector of type `double`: it has to be `double` because its elements are `double`.

```
typeof(ages)
```

```
## [1] "double"
```

We can names the elements of a vector when we create it with `c()`, or we can name them later on. One way to do this is with the `names()` function:

```
names(heights) <- c("Scarecrow", "Tinman", "Lion", "Dorothy", "Toto", "Boq")
heights
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        72        70        69        58        NA        45
```

2.1.7 Special Character Vectors

R comes with two handy, predefined character vectors:

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

We will make use of them from time to time.

2.1.8 Length of Vectors

The `length()` function tells us how many elements a vector has:

```
length(heights)
```

```
## [1] 6
```

2.2 Constructing Patterned Vectors

Quite often we need to make lengthy vectors that follow simple patterns. R has a few functions to assist us in these tasks.

2.2.1 Sequencing

Consider the `seq()` function:

```
seq(from = 5, to = 15, by = 1)
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

The default value of the parameter `by` is 1, so we could get the same thing with:

```
seq(from = 5, to = 15)
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

Further reduction in typing may be achieved as long as we remember the order in which R expects the parameters (`from` before `to`, then `by` if supplied):

```
seq(5, 15)
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

Some more complex examples:

```
seq(3, 15, 2)
```

```
## [1]  3  5  7  9 11 13 15
```

```
seq(0, 1, 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

R will go up to the `to` value, but not past it:

```
seq(3, 16, 2)
```

```
## [1]  3  5  7  9 11 13 15
```

Negative steps are fine:

```
seq(5, -4, -1)
```

```
##  [1]  5  4  3  2  1  0 -1 -2 -3 -4
```

The *colon operator* `:` is a convenient abbreviation for `seq`:

```
1:5 # 1 is from, 5 is to
```

```
## [1] 1 2 3 4 5
```

If the `from` number is greater than the `to` number the step for the colon operator is -1:

```
5:1
```

```
## [1] 5 4 3 2 1
```

2.2.2 Repeating

With `rep()` we may repeat a given vector as many times as we like:

```
rep(3, times = 5)
```

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

We can apply `rep()` to a vector of length greater than 1:

```
vec <- c(7, 3, 4)
rep(vec, times = 3)
```

```
## [1] 7 3 4 7 3 4 7 3 4
```

`rep()` applies perfectly well to character-vectors:

```
rep("Toto", 4)
```

```
## [1] "Toto" "Toto" "Toto" "Toto"
```

`rep()` also takes an `each` parameter that determines how many times each element of the given vector will be repeated before the `times` parameter is applied. This is best illustrated with an example:

```
vec <- c(7, 3, 4)
rep(vec, each = 2, times = 3)
```

```
##  [1] 7 7 3 3 4 4 7 7 3 3 4 4 7 7 3 3 4 4
```

If we combine `seq()` and `rep()` we can create fairly complex patterns concisely:

```
vec <- seq(5, -3, -2)
rep(vec, each = 2, times = 2)
```

```
##  [1]  5  5  3  3  1  1 -1 -1 -3 -3  5  5  3  3  1  1 -1 -1 -3 -3
```

In order to create fifty 10's followed by fifty 30's followed by fifty 50's I would write:

```
rep(seq(10, 50, 20), each = 50)
```

2.3 Subsetting Vectors

Quite often we need to select one or more elements from a vector. The *subsetting operator* `[` allows us to do this.

Recall the vector `heights`:

```
heights
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        72        70        69        58        NA        45
```

If we want the fourth element, we ask for it with the subsetting operator like this:

```
heights[4]
```

```
## Dorothy
##      58
```

If we want two or more elements, then we specify their indices in a vector. Thus, to get the first and fifth elements, we might do this:

```
desired <- c(1,5)
heights[desired]
```

```
## Scarecrow      Toto
##        72        NA
```

We could also ask for them directly:

```
heights[c(1,5)]
```

```
## Scarecrow      Toto
##        72        NA
```

Negative numbers are significant in subsetting:

```
heights[-2] #select all but second element
```

```
## Scarecrow      Lion   Dorothy      Toto       Boq
##        72        69        58        NA        45
```

```
heights[-c(1,3)]  # all but first and third
```

```
##  Tinman Dorothy    Toto     Boq
##      70      58      NA      45
```

If you specify a nonexistent index, you get `NA`, the reasonable result:

```
heights[7]
```

```
## <NA>
##   NA
```

Patterned vectors are quite useful for subsetting. If you want the first three elements of `heights`, you don't have to type `heights[c(1,2,3)]`. Instead you can just say:

```
heights[1:3]
```

```
## Scarecrow    Tinman      Lion
##        72        70        69
```

The following gives the same as `heights`:

```
heights[1:length(heights)]
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        72        70        69        58        NA        45
```

If you desire to quickly provide names for a vector, subsetting can help:

```
vec <- c(23, 14, 82, 33, 33, 45)
names(vec) <- LETTERS[1:length(vec)]
vec
```

```
##  A  B  C  D  E  F
## 23 14 82 33 33 45
```

If a vector has names we can refer to its elements using the subsetting operator and those names:

```
heights["Tinman"]
```

```
## Tinman
##     70
```

```
heights[c("Scarecrow", "Boq")]
```

```
## Scarecrow       Boq
##        72        45
```

Finally, we can use subsetting to modify parts of a vector. For example, Dorothy's height is reported as:

```
heights["Dorothy"]
```

```
## Dorothy
##      58
```

If Dorothy grows two inches, then we can modify her height as follows:

```
heights["Dorothy"] <- 60
```

We can replace more than one element, of course. Thus:

```
heights[c("Scarecrow", "Boq")] <- c(73, 46)
```

The subset of indices may be as complex as you like:

```
vec <- c(3,4,5,6,7,8)
vec[seq(from = 2, to = 6, by = 2)] <- c(100, 200, 300)
vec
```

```
## [1]   3 100   5 200   7 300
```

In the above example, `seq(2,6,2)` identified 2, 4 and 6 as the indices of elements of `vec` that were to be replaced by the corresponding elements of `c(100, 200, 300)`.

We can even use subsetting to rearrange the elements of a vector. Consider the example below:

```
inhabitants <- c("Oz", "Toto", "Boq", "Glinda")
permuted <- inhabitants[c(3,4,1,2)]
permuted
```

```
## [1] "Boq"    "Glinda" "Oz"     "Toto"
```

2.4 More on Logical Vectors

Consider the following expression:

```
13 < 20
```

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

We constructed it with the "less-than" operator `<`. You can think of it as saying that 13 is less than 20, which is a true statement, and sure enough, R *evaluates* the expression `13 < 20` as `TRUE`.

When you think about it, we've seen lots of expressions so far. Here are just a few of them:

- `sqrt(64)`
- `heights`
- `heights[1:3]`
- `13 < 20`

When we type any one of them into the console, it *evaluates* to a particular value. In the examples above, the value was always a vector.

Table 2.1: The Boolean Operators

Operation	What It Means
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
&	and
|	or
&&	and (scalar version)
||	or (scalar version)
!	not

Expressions like `13 < 20` that evaluate to a logical vector are often called *Boolean* expressions.[1]

2.4.1 Boolean Operators

Let's look further into Boolean expressions. Define the following two vectors:

```
a <- c(10, 13, 17)
b <- c(8, 15, 12)
```

Now let's evaluate the expression `a < b`:

```
a < b
```

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

The `<` operator, when applied to vectors, always works *element-wise*; that is, it is applied to corresponding elements of the vectors on either side of it. R's evaluation of `a < b` involves evaluation of the following three expressions:

- `10 < 8` (evaluates to `FALSE`)
- `13 < 15`(evaluates to `TRUE`)
- `17 < 12`(evaluates to `FALSE`)

The result is a logical vector of length 3.

The `<` operator is an example of a *Boolean operator* in R. Table 2.4.1 shows the available Boolean operators.

2.4.1.1 Inequalities

The "numerical-looking operators" (`<`, `<=`, `>`, `>=`) have their usual meanings when one is working with numerical vectors[2] When applied to character vectors they evaluate according to an alphabetical order:

```
a<- c("Dorothy", "toto", "Boq")
b <- c("tinman", "Toto", "2017")
a < b
```

[1] So-called after George Boole, a nineteenth century British logician.
[2] A vector is said to be *numerical* if it is of type `integer` or `double`.

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

The reasons for the evaluation above are as follows:

- D comes before t in the alphabet;
- lowercase t comes before uppercase T, according to R;
- characters for numbers come before letter-characters, according to R.

2.4.1.2 Equality

The equality (==) operator indicates whether the expressions being compared evaluate to the same value. Note that it's made with *two* equal-signs, not one! It's all about evaluation to the same value, not strict identity. The following examples will help to clarify this.

```
a <- c(Dorothy = 1,Toto = 2) # a named vector
b <- c(Glinda = 1, Tinman = 2)
a == b
```

```
## Dorothy    Toto
##    TRUE    TRUE
```

(Note that the resulting logical vector inherits the names of `a`, the vector on the left.).

But `a` and `b` aren't *identical*. We can see this because R has the function `identical()` to test for identity:

```
identical(a, b)
```

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

Corresponding elements of `a` and `b` have the same values, but the two vectors don't have the same set of names, so they aren't considered identical.

Here's another way to see that "evaluating to the same value" is not the same as "identity":

```
TRUE == 1
```

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

When `TRUE (itself oftype logical)` is being compared with something numerical (type `integer` or `double`) it is *coerced* into the numerical vector 1. (In the same situation `FALSE` would be coerced to 0.) But clearly `TRUE` and 1 are not identical:

```
identical(TRUE, 1)
```

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

2.4.1.3 And, Or, Not

We consider an "and" statement to be true when both of its component statements are true; otherwise it is counted as false. The `&` Boolean operator accords with our thinking:

```
a <- c(TRUE, TRUE, FALSE, FALSE)
b <- c(TRUE, FALSE, TRUE, FALSE)
a & b
```

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

In logic and mathematics, an "or" statement is considered to be true when *at least one* of its component statements are true. (This is sometimes called the "inclusive" use of the term "or.") R accords with this line of thinking:

```
a <- c(TRUE, TRUE, FALSE, FALSE)
b <- c(TRUE, FALSE, TRUE, FALSE)
a | b
```

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

The `&&` and `||` operators follow the "and" and "or" logic respectively, but are applied only to the first elements of the vectors being compared:

```
a <- c(TRUE, TRUE)
b <- c(TRUE, TRUE)
a && b
```

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

```
a <- c(TRUE, FALSE)
b <- c(FALSE, FALSE)
a || b
```

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

These operators will come in handy later on, when we study *conditionals*.

The final Boolean operator is `!`, which works like "not":

```
a <- c(TRUE, FALSE)
!a
```

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

```
e <- c(2, 5, 6)
f <-  c(3, 1, 2)
e > f
```

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

```
!(e > f)
```

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

2.5 Vector Recycling

Consider the vector

```
vec <- c(2, 6, 1, 7, 3)
```

Look at what happens when we evaluate the expression:

```
vec > 4
```

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

At first blush this doesn't make any sense: `vec` has length 5, whereas `4` is a vector of length 1. How can the two of them be compared?

They cannot, in fact, be compared. Instead the shorter of the two vectors—the `4`—is *recycled* into the `c(4,4,4,4,4)` a vector of length five, which may then be compared element-wise with `vec`. Recycling is a great convenience as it allows us to express an idea clearly and concisely.

Recycling is always performed on the shorter of two vectors. Consider the example below:

```
vec2 <- 1:6
vec2 > c(3,1)
```

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

Here, `c(3,1)` was recycled into `c(3,1,3,1,3,1)` prior to being compared with `vec2`.

What happens if the length of the longer vector is not a multiple of the shorter one? We should look into this:

```
vec2 <- 1:7
vec2 > c(3, 8)
```

```
## longer object length is not a multiple of shorter object length
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

We get a warning, but R tries to do the job for us anyway, recycling the shorter vector to `c(3,8,3,8,3,8,3)` and then performing the comparison.

By the way, if you don't want to see the warning you can put the expression into the `suppressWarnings()` function:

```
suppressWarnings(vec2 > c(3, 8))
```

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

2.6 Subsetting with Logical Vectors

The subsetting we have seen up to now involves specifying the *indices* of the elements we would like to select from the original vector. It is also possible to say, for each element, *whether or not it is to be included* in our selection. This is accomplished by means of logical vectors.

Recall our `heights` vector:

```
heights
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        73        70        69        60        NA        46
```

Let's say that we want the heights of Scarecrow, Tinman and Dorothy. We can use a logical vector to do this:

```
wanted <- c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)
heights[wanted]
```

```
## Scarecrow    Tinman   Dorothy
##        73        70        60
```

The `TRUE`'s at indices 1, 2, and 4 in `wanted` inform R that we want the heights vector at indices 1, 2 and 4. The `FALSE`'s say: "don't include this element!"

Subsetting can be used powerfully along with logical vectors and Boolean operators.

For example, in order to select those persons whose heights exceed a certain amount, we might say something like this:

```
#heights of some people:
people <- c(55, 64, 67, 70, 63, 72)
tall <- (people >= 70)
tall
```

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

```
people[tall]
```

```
## [1] 70 72
```

As you can see, the `tall` vector specifies which elements we would like to select from the `people` vector.

We need not define the `tall` vector along the way. It is quite common to see something like the following:

```
people[people >= 70]
```

```
## [1] 70 72
```

I like to pronounce the above as:

> *`people`, where `people` is at least 70*

The word "where" in the above phrase corresponds to the subsetting operator.

Your subsetting logical vector need not have been constructed with the original vector in mind. Consider the following example:

```
age <- c(23, 21, 22, 25, 63)
height <- c(68, 67, 71, 70, 69)
age[height < 70]
```

```
## [1] 23 21 63
```

Here the selection is done from the `age` vector, using a logical vector that was constructed from `height`—another vector altogether. It concisely expresses the idea:

> *the ages of people whose height is less than 70*

There is no limit to the complexity of selection. Consider the following:

```
age <- c(23, 21, 22, 25, 63)
height <- c(68, 67, 71, 70, 69)
likesToto <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
height[age < 60 & likesToto]
```

```
## [1] 68 67
```

2.6.1 Counting

Logical subsetting provides a convenient way to *count* the elements of a vector that possess a given property. For example, to find out how many elements of `people` are less than 70 we could say:

```
length(people[people < 70])
```

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

2.6.2 Cautions about NA

You should be aware of the effect of `NA`-values on subsetting.

```
heights
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        73        70        69        60        NA        46
```

```
tall <- (heights > 65)
tall
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##      TRUE      TRUE      TRUE     FALSE        NA     FALSE
```

Since Toto's height was missing, R can't say whether or not he was more than 65 inches tall. Hence it assigns `NA` to the Toto-element of the `tall` vector.

When we subset using this vector we get an odd result:

```
heights[tall]
```

```
## Scarecrow    Tinman      Lion      <NA>
##        73        70        69        NA
```

Since R doesn't know whether or not to select Toto, it records its indecision by including an `NA` in the result. That `NA`, however, is not the `NA` for Toto's height in the vector `heights`, so it can't inherit the "Toto" name. Since it has no name, R presents its name as `<NA>`.

If we try to count the number of tall persons, we get a misleading result:

```
length(heights[tall])
```

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

We would have preferred something like:

> "Three, with another one undecided."

Counting is one those situations in which we might wish to remove `NA` values at the start. If the vector is small we could remove them by hand, e.g.:

```
knownHeights <- heights[-5]  # remove Toto
tall <- (knownHeights > 65)
length(knownHeights[tall])
```

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

For longer vectors the above approach won't be practical. Instead we may use the `is.na()` function.

```
is.na(heights)
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##     FALSE     FALSE     FALSE     FALSE      TRUE     FALSE
```

Then we may select those elements that are *not* `NA`:

```
knownHeights <- heights[!is.na(heights)]
knownHeights
```

```
## Scarecrow    Tinman      Lion   Dorothy       Boq
##        73        70        69        60        46
```

```
length(knownHeights[knownHeights > 65])
```

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

2.6.3 Which, Any, All

There are several functions on logical vectors that are worth keeping in your back pocket:

- `which()`
- `any()`
- `all()`

2.6.3.1 `which()`

Applied to a logical vector, the `which()` function returns the *indices* of the vector that have the value `TRUE`:

```
boolVec <- c(TRUE,TRUE,FALSE,TRUE)
which(boolVec)
```

```
## [1] 1 2 4
```

Thus if we want to know the indices of `heights` where the heights are at least 65, then we write:

```
which(heights > 65)
```

```
## Scarecrow    Tinman      Lion
##         1         2         3
```

(Recall that height was a named vector. The logical vector `heights > 65` inherited these names and passed them on to the result of `whihc()`.)

Note also that Toto's `NA` height was ignored by `which()`.

2.6.3.2 `any()`

Is anyone more than 71 inches tall? `any()` will tell us:

```
heights
```

```
## Scarecrow    Tinman      Lion   Dorothy      Toto       Boq
##        73        70        69        60        NA        46
```

```
any(heights > 71)
```

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

Yes: the Scarecrow is more than 71 inches tall.

We can use `any()` along with the equality Boolean operator `==` to determine whether or not a given value appears a a given vector:

```
vec <- c("Dorothy", "Tin Man", "Scarecrow", "Glinda")
any(vec == "Tin Man")
```

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

```
any(vec == "Wizard")
```

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

The above question occurs so frequently that R provides the `%in%` operator as a short-cut:

```
"Tin Man" %in% vec
```

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

```
"Wizard" %in% vec
```

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

2.6.3.3 `all()`

Is everyone more than 71 inches tall?

Table 2.2: Basic arithmetical operations on vectors.

Operation	What It Means
x + y	addition
x - y	subtraction
x * y	multiplication
x / y	division
x^y	exponentiation (raise x to the power y)
x %/% y	integer division (quotient after dividing x by y)
x %% y	x mod y (remainder after dividing x by y)

```
all(heights > 71)
```

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

2.6.3.4 NA-Caution

Is everyone more than 40 inches tall?

```
all(heights > 40)
```

```
## [1] NA
```

Everyone with a known height is taller than 40 inches, but because Toto's height is `NA` R can't say whether *all* the heights are bigger than 40.

2.7 Basic Arithmetical Operations on Vectors

R provides a number of arithmetical operations on pairs of numerical vectors. Table 2.2 shows the basic operators.

The operators are applied element-wise to vectors:

```
x <- c(10, 15, 20)
y <- c(3, 4, 5)
x + y
```

```
## [1] 13 19 25
```

```
x - y
```

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

```
x * y
```

```
## [1]  30  60 100
```

```
x / y
```

```
## [1] 3.333333 3.750000 4.000000
```

```
x^y
```

```
## [1]    1000   50625 3200000
```

As an illustration, the final result is:

$$10^3, 15^4, 20^5.$$

The "mod" operator %% can be quite useful. Here is an example: even numbers have a remainder of 0 after division by 2, whereas odd numbers have a remainder of 1. Hence we may use %% to quickly locate the even numbers in a vector, as follows:

```
vec <- c(2, 7, 9, 12, 15, 24)
vec[vec %% 2 == 0]
```

```
## [1]  2 12 24
```

Recycling applies in vector arithmetic (as in most of R):

```
vec <- c(2, 7, 9, 12, 15, 24)
2 * vec  # the 2 will be recycled
```

```
## [1]  4 14 18 24 30 48
```

```
vec + 100 # the 100 will be recycled
```

```
## [1] 102 107 109 112 115 124
```

```
vec^3  # the 3 will be recycled
```

```
## [1]     8   343   729  1728  3375 13824
```

2.7.1 More Math Functions

You have already met `sqrt()`. Here are a few more useful math functions involving vectors.

2.7.1.1 Rounding

You can use the `round()` function to round off numbers to any desired number of decimal places.

```
roots <- sqrt(1:5)
roots # Too much information!
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
round(roots, digits = 3) # nicer
```

```
## [1] 1.000 1.414 1.732 2.000 2.236
```

2.7.1.2 Ceiling and Floor

The `ceiling()` function returns the least integer that is greater than or equal to the given number:

```
vec <- c(-2.9, -1.1, 0.2, 1.35, 3, 4.01)
ceiling(vec)
```

```
## [1] -2 -1  1  2  3  5
```

The `floor()` function returns the greatest integer that is less than or equal to the given number:

```
floor(vec)
```

```
## [1] -3 -2  0  1  3  4
```

2.7.1.3 Vectorization

All of the above operations follow the "vector-in, vector-out" principle—often referred to by R users as *vectorization*—to which R often adheres. Not only does vectorization permit us to express ideas concisely and in human-readable fashion, but the computations themselves tend to be performed very quickly.

2.7.1.4 Summing and the Mean

There are some functions on vectors that return only a vector of length 1. Among examples we have met so far are:

- `length()`
- `any()`
- `all()`

Another very important function that returns a vector of length 1 is `sum()` :

```
vecs <- 1:100
sum(vecs)
```

```
## [1] 5050
```

In statistics we are often interested in the *mean* of a list of numbers. The mean is defined as:

$$\frac{\text{sum of the numbers}}{\text{how many numbers there are}}$$

You can find the mean of a numerical vector as follows:

```
vec <- c(-3, 4, 17, 23, 51)
meanVec <- sum(vec)/length(vec)
```

The way we compute the mean in R looks a great deal like its mathematical definition.

You might be interested to know that there is a function in R dedicated to finding the mean. Unsurprisingly, it is called `mean()` :

```
mean(vec)
```

```
## [1] 18.4
```

2.7.1.5 Maximum and Minimum

The `max()` function delivers the maximum value of the elements of a numerical vector:

```
max(c(3, 7, 2))
```

```
## [1] 7
```

The `min()` function delivers the minimum value of a numerical vector:

```
min(c(3, 7, 2))
```

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

You can enter more than one vector into `min()` or `max()`: the function will combine the vectors and then do its job:

```
a <- c(5, 6, 10)
b <- c(2, 3, 12, 15, 1)  # the max of both is 15
max(a, b)
```

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

Both functions yield `NA` when one of the elements is `NA`:

```
max(3, 7, -2, NA)
```

```
## [1] NA
```

Like `sum()` and `mean()`, they respond to the `na.rm` parameter:

```
max(3, 7, -2, NA, na.rm =TRUE)
```

```
## [1] 7
```

The `pmax()` function compares corresponding elements of each input-vector and produces a vector of the maximum values:

```
a <- c(3, 7, 10)
b <- c(5, 2, 12)
pmax(a, b)
```

```
## [1]  5  7 12
```

There is a `pmin()` function that computes pair-wise minima as well.

2.7.2 NA and NaN Considerations

What happens when you are doing mathematics on a vector, one of whose values is `NA`? A vectorizing function will simply pass it along:

```
vec <- c(1, 2, 3, 4, NA)
sqrt(vec)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000       NA
```

On the other hand a function like `sum()` needs to *know* all of the values. If one of them is `NA`, it will report their sum as `NA`.

```
sum(vec)
```

```
## [1] NA
```

The same is true for the mean:

```
mean(vec)
```

```
## [1] NA
```

If we want the sum or the mean of the *known* values, we could first remove the `NA` values as demonstrated in previous sections. We could also make use of the `na.rm` parameter that these functions provide:

```
sum(vec, na.rm = TRUE)
```

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

```
mean(vec , na.rm = TRUE)
```

```
## [1] 2.5
```

The results of some arithmetical operations sometimes are not defined. (Examples: you can't divide by 0; you can't take the square root of a negative number.) R reports the results of such operations as `NaN`—"not a number." R also issues a warning:

```
sqrt(c(-4, 2, 4))
```

```
## Warning in sqrt(c(-4, 2, 4)): NaNs produced
## [1]      NaN 1.414214 2.000000
```

Keep in mind, though, that the result is a perfectly good vector as far as R is concerned. After the warning R will permit you to use it in further computations:

```
vec<- sqrt(c(-4, 2, 4))
```

```
## Warning in sqrt(c(-4, 2, 4)): NaNs produced
```

```
vec + 3
```

```
## [1]      NaN 4.414214 5.000000
```

2.8 Further Notes on Syntax

In the process of learning about R, you have been unconsciously imbibing some of its *syntax*. The *syntax* of a computer-programming is the complete set of rules that determine what combinations of symbols are considered to make a well-formed program in the language—something that R can interpret and attempt to execute.

2.8.1 Syntax Errors vs. Run-time Errors vs. Semantic Errors

For the most part you will learn the syntax informally. By now, for example, you have probably realized that when you call a function you have to supply a closing parenthesis to match the open parenthesis. Thus the following is completely fine:

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

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

On the other hand if you were to type `sum(1:5` alone on a single line in a R script, R Studio's code-checker would show a red warning-circle at that line. Hovering over the circle you would see the message:

```
unmatched opening bracket '('
```

If you were to attempt to run the command `sum(1:5` from the script you would get the following error message:

```
## Error: Incomplete expression: sum(1:5
```

Such an error is called a *syntax error*.[3] The R Studio IDE can detect most—but not all—syntax errors.

Syntax errors in computer programming are similar to grammatical errors in ordinary language, such as:

- "Mice is scary." (Number of the subject does not match the number of the verb.)
- "Mice are." (Incomplete expression.)

A *run-time error* is an error that occurs when the syntax is correct but R is unable to finish the execution of your code for some other reason. The following code, for example, is perfectly fine from a syntactical point of view:

```
sum("hello")
```

When run, however, it produces an error:

```
## Error in sum("hello") : invalid 'type' (character) of argument
```

Here is another example:

```
sum(emeraldCity)
```

Unless for some reason you have defined the variable `emeraldCity`, an attempt to run the above command will produce the following run-time error:

```
## Error: object 'emeraldCity' not found
```

[3] R is a bit more forgiving if you type `sum(1:5` directly into the console and press Enter. Instead of throwing an error, R shows a `+` prompt, hoping for further input that would correctly complete the command. If you are ever in the situation where you do not know how to complete the command, you may simply press the Escape key (upper left-hand corner of your keyboard): R will then abort the command and return to a regular prompt.

Many run-time errors in computer programming resemble errors in ordinary language where the sentence is grammatically correct by does not *mean* anything, as in:

- "Beelbubs are juicy." (What's a "beelbub"?)

There is a third type of error, known in the world of programming as a *semantic error*. The term "semantics" refers to the meaning of things. Computer code is said to contain a semantic error when it is syntactically correct and can be executed, but does not deliver the results one knows to expect.

As an example, suppose you have defined, at some point, two variables:

```
emeraldCity <- 15
emeraldcity <- 4
```

Suppose now that—wanting R to compute 15^2—you run the following code:

```
emeraldcity^2
```

```
## [1] 16
```

You don't get the results you wanted, because you accidentally asked for the square of the wrong number.

Semantic errors are usually the most difficult errors for programmers to detect and repair.

2.8.2 The Assignment Operator

We have been using the assignment operator `<-` to assign values to variables. You should be aware that there is another assignment operator that works the other way around:

```
4 -> emeraldCity
emeraldCity
```

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

Most people don't use it.

A popular alternative to `<-` as an assignment operator is the equals sign `=`:

```
emeraldCity = 5
emeraldCity
```

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

I myself prefer to stay away from it, as it can be confused with other uses of `=`, such as the setting of values to parameters in functions:

```
rep("Dorothy", times = 3)
```

```
## [1] "Dorothy" "Dorothy" "Dorothy"
```

When you have to assign the same value to several values, R allows you to abbreviate a bit. Consider the following code:

```
a <- b <- c <- 5
```

The above code has the same effect as:

```
a <- 5
b <- 5
c <- 5
```

2.8.3 Multiple Expressions

R allows you to write more than one expression on a single line, as long as you separate the expressions with semicolons:

```
a <- b <- c <- 5
a; b; c; 2+2; sum(1:5)
```

```
## [1] 5
## [1] 5
## [1] 5
## [1] 4
## [1] 15
```

2.8.4 Variable Names and Reserved Words

Using the assignment operator we have created quite a few variables by now, and we appear to have named them whatever we want. In fact there are very few limitation on the name of a variable. According to R's own documentation:[4]

> "A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number."

This leaves a lot of room for creativity. All of the following names are possible for variables:

- `yellowBrickRoad`
- `yellow_brick_road`
- `yellow.brick.road`
- `yell23`
- `y2e3L45Lo3....rOAD`
- `.yellow`

The following, though, are not valid:

- `.2yellow` (cannot start with dot and then number)
- `_yellow` (cannot start with `_`)
- `5scones` (cannot start with a number)

Most programmers try to devise names for variables that are *descriptive* in the sense that they suggest to a reader of the code the role that is played within it by the variable. In addition they try to stick to a consistent system for variable names that divide naturally into meaningful words.

One popular convention is known as CamelCase. In this convention each new word-like part of the variable names begins with a capital letter. (The initial letter, though, is often not capitalized.) Examples would be:

[4] See `help(make.names)`.

- `emeraldCity`
- `isEven`

Another popular convention—sometimes called "snake-case"—is to use lowercase and to separate words with underscores:

- `emerald_city`
- `is_even`

An older convention—one that was popular among some of the original developers of R—was to separate words with dots:

- `emerald.city`
- `is.even`

This last convention is no longer recommended, as in programming languages other than R the dot is associated syntactically with the calling of a "method" on an "object."[5]

There is one further restriction on variable-names that we have not yet mentioned: you are not allowed to use any of R's *reserved words*. These are:

> `if`, `else`, `while`, `repeat`, `function`, `for`, `in`, `next`, `break`,`TRUE`, `FALSE`, `NULL`, `inf`, `NaN`, `NA`, `NA_integer`, `NA_real`, `NA_complex`, `NA_character`

You need not memorize the above list: You'll gradually learn most of it, and words you don't learn are words that you are unlikely to ever choose as a variable-name on your own. Besides, reserved words show in in blue in the R Studio editor, and if you manage to use one anyway then R will stop you outright with a clear error message:

```
break <- 5
```

```
## Error in break <- 5 : invalid (NULL) left side of assignment
```

Notice also that although `TRUE` and `FALSE` are reserved words, their accepted abbreviations `T` and `F` are not. This can lead to problems in code, if someone chooses to bind `T` or `F` to some value.

For example, suppose that have two lines of code like this:

```
T <- 0
F <- 1
```

Later on, suppose you create what you think is a logical vector:

```
myVector <- c(T, F, F, T)
```

But it's *not* logical:

```
typeof(myVector)
```

```
## [1] "double"
```

That's because `T` ad `F` have been bound to numerical values. If you coerce `myLogical` to a logical vector, you get the exact opposite of what you would have expected:

```
as.logical(myVector)
```

[5] We will look briefly at R's object-oriented capabilities in Chapter 14.

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

The moral of the story is:

To be safe, never use `T` for `TRUE` or `F` for `FALSE`.

One final remark: variables together with reserved words constitute the part of the R language called *identifiers*.

Glossary

Vector Type Any one of the six basic forms the elements in an atomic vector can take. The four types we will encounter the most are: double, integer, character and logical.

Coercion The process of changing a vector from one type to another. Sometimes the process takes place automatically, as a convenience to the programmer.

Sub-setting The operation of selecting one or more elements from a vector.

Recycling An automatic process by which R, when given two vectors, repeats elements of the shorter vector until it is as long as the longer vector. Recycling enables the two resulting vectors to be combined element-wise in operations.

Vectorization R's ability to operate on each element of a vector, producing a new vector of the same length. Vectorized operations can be expressed concisely and performed very quickly.

Reserved Words Identifiers that are set aside by R for specific programming purposes. They cannot be used as names of variables.

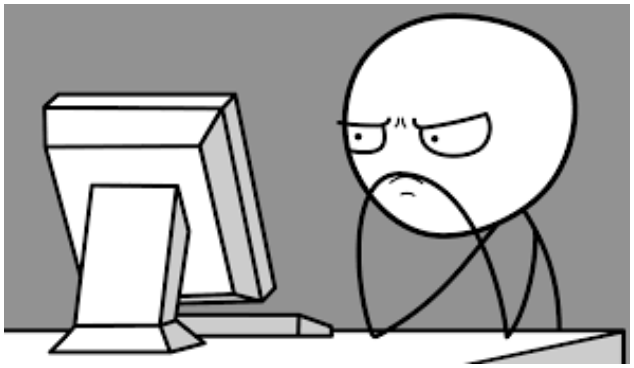
Syntax The complete set of rules for a computer language that determine what combinations of symbols are considered to make a well-formed program in the language.

Syntax Error A sequence of symbols that contains a violation of one of the rules of syntax. R is unable to interpret and attempt to execute code that contains a syntax error.

Run-time Error An error that occurs when the computer language's interpreter attempts to execute code but is unable to do so. A typical cause of a run-time error is the situation when the code calls for the evaluation of a name that has not been bound to an object.

Semantic Error An error in code that is syntactically correct and that can be executed by the computer but which produces unexpected results.

Exercises



1. Determine the type of each of the following vectors:
    1. `c(3.2, 2L, 4.7, TRUE)`
    2. `c(as.integer(3.2), 2L, 5L, TRUE)`
    3. `c(as.integer(3.2), 2L, "5L", TRUE)`
2. Using a combination of `c()`, `rep()` and `seq()` and other operations, find concise one-line programs to produce each of the following vectors:
    1. all numbers from 4 to 307 that are one more than a multiple of 3;
    2. the numbers 0.01, 0.02, 0.03, ..., 0.98, 0.99.
    3. twelve 2's, followed by twelve 4's followed by twelve 6's, ..., followed by twelve 10's, finishing with twelve 12's.
    4. one 1, followed by two 2's, followed by three 3's, ..., followed by nine 9's, finishing with ten 10's.
3. The following three vectors gives the names, heights and ages of five people, and also say whether or not each person likes Toto:

```
person <- c("Akash", "Bee", "Celia", "Devadatta", "Enid")
age <- c(23, 21, 22, 25, 63)
height <- c(68, 67, 71, 70, 69)
likesToto <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
```

    Use sub-setting with logical vectors to produce vectors of:
    1. the names of all people over the age of 22;
    2. the names of all people younger than 24 who are also more than 67 inches tall;
    3. the names of all people who either don't like Toto or who are over the age of 30;
    4. the *number* of people who are over the age of 22.
4. Logical vectors are not numerical vectors, so it would seem that you should not be able to sum their elements. But:

```
sum(likesToto)
```

    results in the number 2! What is happening here is that R coerces the logical vector `likesToto` into a numerical vector of 1's and 0's—1 for `TRUE`, 0 for `FALSE`—and then sums the resulting vector. Notice that this gives us the number of people who like Toto. With this idea in mind, use `sum()` along with logical vectors to find:
    1. the number of people younger than 24 who are also more than 67 inches tall;
    2. the number of people who either don't like Toto or who are over the age of 30.

Chapter 3

Functions

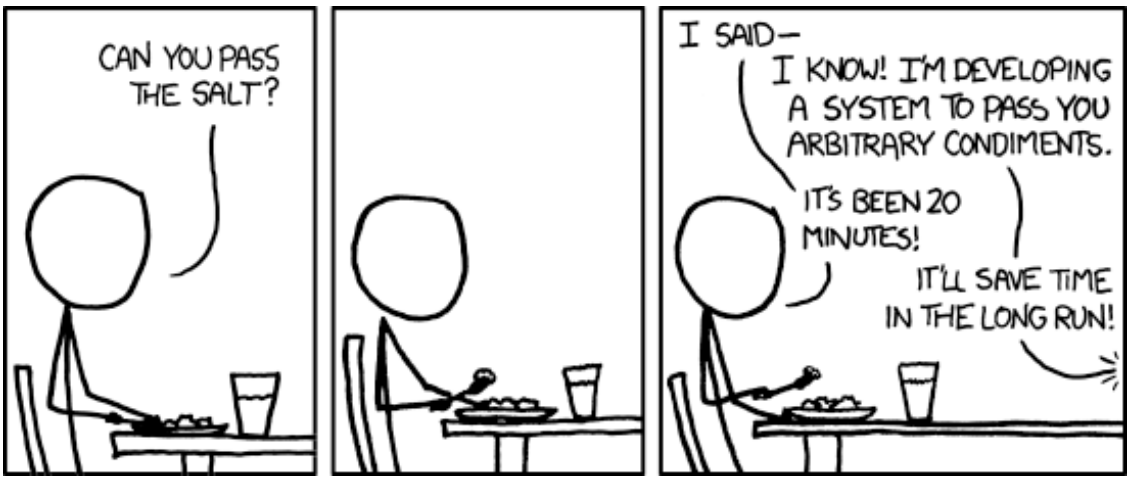


Figure 3.1: The General Problem, by xkcd.

The xkcd cartoon alludes to a common aspiration of programmers: to solve a frequently-occurring problem *in general* so that we don't have to keep on devising solutions specific for each case. In R and in most other programming languages, *functions* are one of the important tools for solving problems in a general manner. In this Chapter we take a close look at how functions work in R. Along the way we'll learn about environments and scoping, receiving input from the user, and a few more built-in R-functions.

3.1 Motivation for Functions

Suppose you have the job of printing out the word "Kansas" to the console four times, each time on a new line. The code for this is easy enough:

```
cat("Kansas\n")
```

```
## Kansas
```

```
cat("Kansas\n")
```

```
## Kansas
```

```
cat("Kansas\n")
```

```
## Kansas
```

```
cat("Kansas\n")
```

```
## Kansas
```

Now suppose that you have the job of printing out *any given* word to the console four times. You could of course, simply copy and paste the above code to a new place in your R script and then change "Kansas" to whatever the desired word is. But that's an awful lot of work.

You could cut down on the work a bit if you use a variable:

```
word <- "Kansas"
cat(word, "\n", sep = "")
cat(word, "\n", sep = "")
cat(word, "\n", sep = "")
cat(word, "\n", sep = "")
```

The advantage of this approach is that, after you copy and paste you only have to make one change, i.e.: substitute the desired word in place of "Kansas" in the assignment to the variable `word`.

If you were writing a program that involved many four-line print-outs of various words, then you could carry on this way quite a while, producing many similar five-line snippets of printing-code throughout your program.

But suppose that it occurs to you one day: maybe you don't really need five lines of code. What if `cat()` supports "vector in, vector out"? If so then we could take advantage of vectorization to obviate the need to repeated calls to 'cat().

We could try:

```
fourWords <- rep("Kansas", 4)
cat(fourWords, "\n")
```

```
## Kansas Kansas Kansas Kansas
```

That just repeats "Kansas" four times, with the default space between in each one—then newline is appended. So we need a newline along with each instance of Kansas.

So instead we try:

```
fourWords <- rep("Kansas\n", 4)
cat(fourWords)
```

```
## Kansas
##  Kansas
##  Kansas
##  Kansas
```

Not quite what we wanted: `cat()` inserts the default space at the end of each instance of `Kansas\n`, resulting in the indentation of lines, 2, 3 and 4.

No problem—let's just set the separation to the empty string "":

```
fourWords <- rep("Kansas\n", 4)
cat(fourWords, sep = "")
```

```
## Kansas
## Kansas
## Kansas
## Kansas
```

Success at last!

If you wanted to implement this new idea throughout your program, you would have to search through the program for the many five-line snippets you created previously, replacing each one of them with the appropriate version of your clever one-liner. Not only is this a lot of work, it's also quite error-prone: you could miss or more of the snippets along the way, or on some occasion fail to modify the word within the one-liner to have the value you need at that point.

Accordingly programmers try, as much as possible, to solve problems in a *general* way and to implement that general solution in *one place* in their program. Then they call upon that solution in the many different locations where the solution might be required.

Functions are one way in which programmers accomplish this. The following is a function that will print any given word four times, once on each line:

```
catFourTimes <- function(word) {
  wordWithNewline <- paste(word, "\n", sep = "")
  cat(rep(wordWithNewline, 4), sep = "")
}
```

Let's see the function in use:

```
catFourTimes("Kansas")
```

```
## Kansas
## Kansas
## Kansas
## Kansas
```

It works like a charm! What's more, once we get to thinking in terms of general solutions, we realize that we might just as well have our function print not only any given word, but print it any given number of times. So instead of `catFourTimes()` we might actually use the following:

```
manyCat <- function(word, n) {
  wordWithNewline <- paste(word, "\n", sep = "")
  lines <- rep(wordWithNewline, times = n)
  cat(lines, sep = "")
}
```

Does it work? Let's see:

```
manyCat("Kansas", 5)
```

```
## Kansas
## Kansas
## Kansas
## Kansas
## Kansas
```

Yes indeed!

Let's consider the advantages of writing functions:

- Functions allow us to re-use code, rather than repeating the code throughout our program.
- The more generally the functions solves the problem, the varied are the situations in which the function may be re-used.
- If we have to change our our approach to the problem—because our original solution was flawed or if there is a need to add new features to our solution, or for any other reason—then we only have to implement the necessary change in the definition of our function, rather than in the many places in the program where the function is actually used.

There is a well-known principle in computer programming called DRY, which is an acronym for *"Don't Repeat Yourself."* Computer code is said to be DRY when general solutions are defined in one place but are usable in many places, and when information needed in many places is defined authoritatively in one place. As a rule, DRY code is easy to develop, debug, read and maintain. The more you get into the habit of expressing solutions to problems in terms of functions, the "drier" your code will be.

3.2 Function Syntax

`function` is the reserved word in R that permits us to define functions. The general form of a function definition is as follows:

```
functionName <- function(parameter, parameter, ...) {
  Body of the Function ...
}
```

`functionName` is the variable that will refer to the function you define. Like any other identifier, it can contain letters, numbers, the dot and the underscore character, but is not permitted to begin with the dot followed by a number.

After the `function` reserved word we see a pair of matching parentheses. They contain the *parameters* of the function, which will be passed into the function as variables referred to by the parameter names.

The *body* of the function consists of one or more expressions that do the work of the function. Note that the body is enclosed with curly braces. This is only necessary, though, if the body consists of more than one

expression. If the body had only one expression then that expression could appear without the braces, like this:

```
add3 <- function(x) x+3
add3(x = 5)
```

```
## [1] 8
```

```
add3(-7)
```

```
## [1] -4
```

In the `add3()` function above, `x` was a parameter. When the function is called the parameter is assigned a particular value called an *argument*. We see that `add3()` was called twice, once with an argument of 3 and again with an argument of -7. If the parameter is explicitly mentioned in the function call, then an equal-sign `=` separates the parameter and the argument. Note also that the parameter and `=` may be omitted if it is clear what parameter the argument will be matched to. In the case of `add3` there is only one parameter `x`, so R knows that any value provided within the parentheses is to be assigned to `x`. R also knows to refer to the order of parameters within the function's definition to determine which arguments go with which parameters. Thus, the following calls do the same thing:

```
manyCat(word = "Hello", n = 4)
manyCat(word = "Hello", 4)
manyCat("Hello", n = 4)
manyCat("Hello", 4)
manyCat(n = 4, word = "Hello")
```

On the other hand, the following would produce an error:

```
manyCat(4, "Hello")
```

```
## NAs introduced by coercion
## Error in rep(wordWithNewline, times = n) :
## invalid 'times' argument
```

If you don't label your arguments with the parameters they are to match to, then you must at least write them in the order in which the parameters appear in the definition of the function.

In the definition of functions and in all calls to functions, commas must separate arguments. Thus the following would produce an error:

```
manyCat("Hello" 4)
```

```
## Error: unexpected numeric constant in "manyCat("Hello" 4"
```

If R cannot match all your arguments to a parameter it will throw an error;

```
manyCat("Hello", 4, 7)
```

```
## Error in manyCat("Hello", 4, 7) : unused argument (7)
```

You will have noticed by now that parentheses are essential when using a function. What would happen if we typed just the function's name itself? Give it a try:

```
manyCat
```

```
## function(word, n) {
##   wordWithNewline <- paste(word, "\n", sep = "")
##   lines <- rep(wordWithNewline, times = n)
##   cat(lines, sep = "")
## }
```

What's printed to the screen is the code that defines the function. The function itself is not called.

3.3 What a Function Returns

In this section we learn about return-values of functions.

3.3.1 The Final Expression Evaluated

Let's write a small function to raise a number to a power:[1]

```
pow <- function(x,y) {
  x^y
}
```

Check to see that it works:

```
pow(2,3)
```

```
## [1] 8
```

All seems well.

If we like we can assign the result of `pow()` to some variable, for use later on:

```
a <- pow(2,4)
cat("I have", a, "cats.")
```

```
## I have 16 cats.
```

In computer programming parlance, `pow(x, y)` is said to *return* the numerical value `x^y`: `pow(2,3)` returns 8, `pow(2,4)` returns 16, and so on.

In R, what a function returns is: *the value of the final expression that it evaluates.* You can see this principle at work in the following example:

```
f <- function(x) {
  2*x + 3
  45
  "hello"
  x^2
}
```

[1] I know, I know—R already has the exponentiation operator. We just need an example to work with, here.

```
f(4)
```

```
## [1] 16
```

We put in 4 as the argument for the parameter `x`, but:

- we did not get back 11 ($2 \times 4 + 3$),
- nor did we get back 45,
- nor did we get back the string "hello".

When `f()` was called, R evaluated all of the expressions in its body, but returned only the value of the final expression it evaluated: $4^2 = 16$.

3.3.2 The `return()` Function

R does have a special function to force a function to cease evaluation at a specific point. Its name, unsurprisingly, is `return()`. Here is an example:

```
g <- function(x) {
  val <- 3*x +7
  return(val)
  "Hello!"
}

g(1)
```

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

We get $3 * 1 + 7 = 10$, but we don't get "Hello!". After returning the 10, the function stopped evaluating expressions: hence it never even bothered to evaluate "Hello", much less to display it in the console.[2]

It follows that it does not matter whether or not you wrap the final expression of a function in `return()`. The following two functions do exactly the same thing:

```
f1 <- function(x) x^2
f2 <- function(x) return(x^2)
```

Some people—especially those who are familiar with other programming languages where return statements are required—like to wrap the final expression in `return()`, simply as a matter of clarity.

3.3.3 Writing a "Talky" Function

Suppose that you would like your function to raise a number to a power, returning the answer to the user, but you also want it to print out a message to the console. You might try writing your function like this:

```
talkySquare <- function(x) {
  result <- x^2
  result
```

[2] You might wonder why anyone would write a function that contains expressions after a call to `return()`. We'll learn why in Chapter 4.

```
  cat("The square of ", x, " is:  ", result, ".\n", sep = "")
}
```

We try it out:

```
talkySquare(4)
```

```
## The square of 4 is:  16.
```

All seems well. But what if we want to save the result in a variable, so that we could perhaps add a number to it later? Something like this, perhaps:

```
a <- talkySquare(4)
```

```
## The square of 4 is:  16.
```

```
a + 4
```

```
## numeric(0)
```

The results don't really make sense. What happened, of course is that R dutifully returned the value of the final expression in the function's body—the result of the `cat()` call, not the value of the variable `result`. If we want both the print-out *and* the square to be returned, the we have to write our functions like this:

```
talkySquare <- function(x) {
  result <- x^2
  cat("The square of ", x, " is:  ", result, ".\n", sep = "")
  result
}
```

This works out as expected:

```
a <- talkySquare(4)
```

```
## The square of 4 is:  16.
```

```
a + 4
```

```
## [1] 20
```

Well, maybe it doesn't work *exactly* as we would like. It would nice if the function would talk to us only when we ask for the results in the console, not when we are simply assigning the results to a variable for later use. In Chapter 4 we will learn how to make our talky function keep quiet when we prefer silence.

3.3.4 The `print()` Function

Consider the following function:

```
grumpySquare <- function(x) {
  "OK, OK, I'm getting to it ... "
  x^2
}
```

We know by now not to expect to see the grumpy message:

```
grumpySquare(4)
```

```
## [1] 16
```

If we want to see the message, we could wrap it in `cat()`. Another possibility is to use the `print()` function:

```
grumpySquare <- function(x) {
  print("OK, OK, I'm getting to it ... ")
  x^2
}
grumpySquare(4)
```

```
## [1] "OK, OK, I'm getting to it ... "
```

```
## [1] 16
```

When R executes a call to `print()` it is forced to print something out to the console, even if it is in the midst of evaluating expressions in a function. The `print`-statement is not involved in what the function returns—that's all up to the expression `x^2`—but it does cause a result *outside of the function itself*. Any external result produced by a function (other than what the function returns) is called a *side-effect* of the function. `cat()` and `print()` are examples of functions that, when called inside of some other function, produce side-effects.

You should know that in R you have been calling the `print()` function quite a bit, without even knowing it. Consider the following line of code:

```
2+2
```

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

R evaluates the expression `2+2`, arriving at the value 4. But what makes the `4` appear on our console? Behind the scenes, R actually evaluated the expression

```
print(2+2)
```

That's what got the 4 into the console!

At this point we don't use `print()` explicitly very much—we just rely on R to call it for us when we are evaluating expressions at the console. Later on we will find that it has other uses.[3]

[3] R is one of very few major programming languages that engage in behind-the-scenes calls to a print function. In many other languages you have to call its print-function explicitly if you want the value of an expression to be displayed.

3.4 More About Arguments

3.4.1 Default Arguments

Sometime around the year 1400CE, the South Indian mathematician Madhava discovered the following infinite-series formula for π, the ratio of the circumference to the diameter of a circle:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

The numerator of each fraction is always 4. The denominators are the odd numbers 1, 3, 5, and so on. The fractions alternate between positive and negative. The idea is that the further you go out in the series, the closer the sum of the fractions will be to π. No matter how close you want to get to π, you can get that close by adding up sufficiently many of the fractions.

In mathematics courses we learn to write the sum like this:

$$\pi = \sum_{k=1}^{k=\infty} (-1)^{k+1} \frac{4}{2k-1}.$$

Here's how the mathematical notation works:

- The Σ sign stands for "sum": it means that we plan to add up a lot of terms.
- The expression $(-1)^k \frac{4}{2k-1}$ after the sum-sign stands for all of the terms that will be added up to make the infinite series.
- Underneath the sum-sign, $k = 1$ says that in the expression after the sum sign we will start by letting k be 1.
- If we let $k = 1$, then the expression becomes

  $$(-1)^2 \frac{4}{2 \cdot 1 - 1} = \frac{4}{1} = 4,$$

  the first term in the series.
- If we let $k = 2$, then the expression becomes

  $$(-1)^3 \frac{4}{2 \cdot 2 - 1} = -\frac{4}{3}$$

  the second term in the series.
- If we let $k = 3$, then the expression becomes

  $$(-1)^4 \frac{4}{2 \cdot 3 - 1} = \frac{4}{5}$$

  the third term in the series.
- The $k = \infty$ above the sum-sign says that we are to keep on going like this, increasing k by 1 every time, without stopping.
- In this way we get the entire infinite series.

What Madhava discovered was that if you *do* stop after some large number of terms, then the sum of the terms you have constructed will be close to π. The more terms you add up before stopping, the closer to π you will get.

Let's write a function to compute the sum of the first n terms of the series, where n can be any value we choose.

```
# function to approximate pi with Madhava's series
# (sum the first n terms of the series)
madhavaPI <- function(n) {
  # make a vector of all of the k's we need:
  k <- 1:n
  # make a vector of the first n terms of the sum:
  terms <- (-1)^(k+1)*4/(2*k-1)
  # return the sum of the terms:
  sum(terms)
}
```

R's vectorization capabilities make it easy to write the code; in fact, the code is essentially a copy of the sum-formula.

Note also the presence of *comments* in the code above. Anything that appears on a line after the pound-sign `#` will be ignored by R. We therefore use the `#`-sign to insert ordinary-language comments into our code in order to explain to others (and to ourselves when we look at the code much later) what we are doing and why we are doing it.

Let's try it out by adding the first million terms of the series:

```
madhavaPI(1000000)
```

```
## [1] 3.141592
```

How close is this to π? R has a built-in constant `pi` that will tell us:

```
pi
```

```
## [1] 3.141593
```

Madhava's approximation was pretty close, although we did have to add up quite a few terms!

The `madhavaPI()` function has a parameter `n` that stands for the number of terms we want to add up. If we don't provide a value for `n`, then there will be an error:

```
madhavaPI()
```

```
## Error in madhavaPI() : argument "n" is missing, with no default
```

There is a way, though, to allow a user to avoid providing a value for `n`. We simply provide a *default value* for it, like this:

```
madhavaPI <- function(n = 1000000) {
  k <- 1:n
  terms <- (-1)^(k+1)*4/(2*k-1)
  sum(terms)
}
```

Now we can call the function without having to specify an argument for the parameter `n`:

```
madhavaPI()
```

```
## [1] 3.141592
```

The function used the default value of 1000000, so it summed the first million terms.

On the other hand, if the user provides his or her own value for `n` then the function will *override* the default:

```
madhavaPI(100)   # only the first hundred terms, please!
```

```
## [1] 3.131593
```

If you are writing a function for which some of the parameters will often be assigned particular values, it would be a kindness to your users to write in these common values as defaults.

3.4.2 Argument-Matching

Sometimes you will see functions in R that provide what appears to be a *vector* of default values. You'll see this in the following example, which concerns a function that uses a named vector to report the favorite color of each of the major groups of inhabitants of the Land of Oz.

```
inhabitants <- c("Munchkin", "Winkie",
            "Quadling", "Gillikin")
favColor <- c("blue", "yellow", "red", "purple")
names(favColor) <- inhabitants

favColorReport <- function(inhabitant = inhabitants) {
  x <- match.arg(inhabitant)
  cat(favColor[x],"\n")
}
```

Here are a couple of sample calls:

```
favColorReport("Winkie")
```

```
## yellow
```

```
favColorReport("Quadling")
```

```
## red
```

It might get tiresome to type out the full name of each group of inhabitants. What would happen if we got a bit lazy?

```
favColorReport("Win")
```

```
## yellow
```

```
favColorReport("Wi")
```

```
## yellow
```

```
favColorReport("W")
```

```
## yellow
```

```
favColorReport("Qua")
```

```
## red
```

```
favColorReport("Gil")
```

```
## purple
```

The key to this behavior is two-fold:

- The vector `inhabitants` was set as the "default value" of the parameter `inhabitant`.
- We called the `match.arg()` function, which found the element of `inhabitants` that matched what the user actually submitted for the parameter `inhabitant`. This element was then assigned to `x`, and we used `x` to report the desired color.

Sometimes when you look at R-help you'll see the default value for a parameter set as a vector—usually a character-vector, as in our example. Most likely what is going on is that somewhere inside the function R will attempt to match the argument you provide with one of the elements of that "default" vector. When a function is written in this way, the possible parameters can have quite long names but the user doesn't have to type them in all the way, as long as the user types enough characters to pick out uniquely an element of the default vector.

The matching is done by exact match of initial characters. For example, it won't do for me to enter:

```
favColorReport("w")  # wants Winkies, but using lower-case w
```

```
## Error in match.arg(inhabitant) : 'arg' should be one of
## "Munchkin", "Winkie", "Quadling", "Gillikin"
```

Note that the default vector isn't really a default *value* for the parameter. Its first element does, however, serve as the default value:

```
favColorReport()  # defaults to "Munchkin"
```

```
## blue
```

3.5 Environments and Scope

3.5.1 Environments and Searching

In R an *environment* is a particular kind of data structure that helps the computer connect names to a value. An environment can be thought of as a bag of names—names for vectors, functions, and all sorts of objects—along with a way (provided automatically by the computer) of getting from each name to the value that it represents. The process that the computer follows in order to connect a name to a value is called *scoping*. In R, as in many other computer languages, environments are what makes scoping possible. In other words, environments are how R figures out what the names in any piece of code *mean*.

R has a considerable number of environments—and environments can be created and destroyed throughout the course of an R session—but at any moment only one of them is *active*. The *active environment* is the first environment that R will examine when it needs to look up a name in an expression.

The most familiar environment is the *Global Environment*—the one that is active when you are using R from the console. The names in the Global Environment, along with descriptions of the objects to which they

refer, are shown in the **Environment** panel in the R Studio IDE. Alternatively, you see the names in the active environment by using the `ls()` function:

```
ls()
```

Even better is `ls.str()`, which will give the name of each object along with a summary of what sort of object it is.

```
ls.str()
```

You can remove all of the names from your Global Environment by pressing the Broom icon in the IDE, or by using the `rm()`function:

```
rm(list = ls())
```

As we mentioned previously, the Global Environment is only one of many environments that exist in R. The `search()` function will show you a number of other environments:

```
search()
```

```
##  [1] ".GlobalEnv"             "package:bindrcpp"
##  [3] "package:R6"             "package:readr"
##  [5] "package:tigerData"      "package:tigerstats"
##  [7] "package:abd"            "package:mosaic"
##  [9] "package:Matrix"         "package:ggformula"
## [11] "package:dplyr"          "package:lattice"
## [13] "package:nlme"           "package:TurtleGraphics"
## [15] "package:grid"           "package:ggplot2"
## [17] "package:mosaicData"     "tools:rstudio"
## [19] "package:stats"          "package:graphics"
## [21] "package:grDevices"      "package:utils"
## [23] "package:datasets"       "package:methods"
## [25] "Autoloads"              "package:base"
```

`search()` returns a character vector of names of environments. The first element is the Global Environment itself. The second element is an environment that is associated with last *package* that R loaded, the third is an environment associated with the next-to-last package, and so on. Each item on the list is considered to be the *parent* of the environment that came before it. Thus, the Global Environment has a parent environment, a grandparent environment, and so on. The complete sequence of environments is called the *search path*.

Just as the Global Environment has names for objects, so also the packages have names available for use. When you write code that contains a name, R will search for that name: first in your Global Environment, then in its parent environment—the environment of the last package loaded—and so on until it reaches the final package on the list: package **base**. If it can't find the name anywhere, then it will throw an error, telling you that the object "cannot be found."

Let's try this with a few examples. First, define a (hopefully) new variable:

```
quadlingColor <- "red"
```

Then use it in some code:

```
cat(quadlingColor, ", white and blue\n", sep ="")
```

```
## red, white and blue
```

R was able to complete your request because:

- it found the name ‘quadlingColor on its search path;
- it found the name `cat` on its search path (and found that it referred to the `cat()` function)

You can tell where R found these things:

```
find("quadlingColor")
```

```
## [1] ".GlobalEnv"
```

```
find("cat")
```

```
## [1] "package:base"
```

R found `quadlingColor` in the first place it looked, whereas it had to go all the way up to package **base** to find an object with the name `cat` that looked like it was the name of a function.

What happens if the same name gets used in two different environments? Let's investigate. First get a print of `cat()`:

```
cat
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## {
##     if (is.character(file))
##         if (file == "")
##             file <- stdout()
##         else if (substring(file, 1L, 1L) == "|") {
##             file <- pipe(substring(file, 2L), "w")
##             on.exit(close(file))
##         }
##         else {
##             file <- file(file, ifelse(append, "a", "w"))
##             on.exit(close(file))
##         }
##     .Internal(cat(list(...), file, sep, fill, labels, append))
## }
## <bytecode: 0x10e6fd8a8>
## <environment: namespace:base>
```

I got the definition of the `cat()` function, all the way up in package **base**.

Now try:

```
rep(cat, times = 3)
```

```
## Error in rep(cat, times = 3) :
##   attempt to replicate an object of type 'closure'
```

I got an error! That's because the only reference R could find for `cat` was to the function `cat()` in package **base**, and since a function isn't a vector you can't repeat it.[4]

Next, define a variable named `cat`:

```
cat <- "Pippin"
```

At this point, we the identifier `cat` appears in at least two environments:

- in the Global Environment, where is refers to the string "Pippin";
- in the environment associated with package **base**, where is refers to the `cat()`-function.

We can verify the above assertions with `find()`:

```
find("cat")
```

```
## [1] ".GlobalEnv"   "package:base"
```

Now try:

```
rep(cat, times = 3)
```

```
## [1] "Pippin" "Pippin" "Pippin"
```

This time it worked! The reason is that R found a character-vector named `cat` in the Global Environment.

Now try:

```
cat(cat, "is a cat\n")
```

```
## Pippin is a cat
```

Wait a minute: why did this work? Doesn't the Global Environment come before package **base** in the search path? Yes it does, but since the first occurrence of `cat` was followed by an open parenthesis R know to expect that it referred to a function. Hence it kept looking along the search path for a function with the name `cat`, eventually finding our familiar `cat()` function in **base**.

Well then, consider happens if we do this:

```
cat <- function(...) {
  "Meow!"
}
```

We have defined a function `cat()` that returns "Meow!" no matter what it is given as input.[5]

Now try again:

```
cat(cat, "is a cat\n")
```

```
## [1] "Meow!"
```

Since the `cat()` we defined is a function in the Global Environment—which comes before **base** in the search path—R uses our `cat()` instead of the **base**'s `cat()`. R programmers say that the **base** version of cat has been *masked*.

[4] In R, almost all functions are called "closures."

[5] The ellipses, which we will discuss further in Chapter 9, allow the function to be passed any arguments at all—or even none.

If I want to keep my `cat()` and still use the **base** version of`cat()` as well, I can do that. In order to be sure of getting a particular package's version of a function, put the name of the package and then two semicolons before the function-name, like this:

```
base::cat("This is the good ol' cat() we have been missing!")
```

```
## This is the good ol' cat() we have been missing!
```

But we don't like our `cat()` so very much: let's remove it:

```
rm(cat)
```

The vector `cat` is removed as well by the previous command.

3.5.2 A Note on Parameters vs. Arguments

It's important to keep in mind the distinction between a *parameter* of a function on the one hand and, on the other hand the *argument* that gets supplied to that parameter.

When you are just starting out in R programming, this distinction can be difficult to remember, especially when the parameter and the argument have the *same name*. Now that we understand environments, though, we can get a grip on this tricky situation.

Let's proceed by way of an example.

First of all, clear out your Global Environment:

```
rm(list = ls())
```

Next make a simple function that adds three to any given number. Our function will take one parameter `n` (the number to which 3 is to be added), and the default value of `n` shall be 4.

```
addThree <- function(n = 4) {
  n + 3
}
```

Next, bind the name `n` to the value 2:

```
n <- 2
```

You should now have two items in your Global Environment. Confirm this:

```
ls.str()
```

```
## addThree : function (n = 4)
## n :  num 2
```

Now call the function as follows:

```
addThree(n = 5)
```

```
## [1] 8
```

Let's recall how this works:

- R sees that you want to assign the value 5 to the parameter `n`.
- R executes the code in the body of the function. All is well.

Now call the function as follows:

```
addThree()
```

```
## [1] 7
```

Let's recall how this works:

- R sees that you did not assign anything to the parameter `n`.
- "That's OK", says R. "I'll use the default value of 4 for `n`."
- R executes the code in the body of the function. All is well.

Next, call the function as follows:

```
addThree(n = n)
```

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

Let's think about how this works:

- R sees that you want to assign something to the parameter `n`. Apparently it is the value of a name `n` in some environment.
- "Fine", says R. "I'll look up the value of this `n` thingie, if ever I have to use it in a computation."
- R executes the one line of code in the body of the function.
- "Well, I'll be darned," says R, "I *do* need the value of this `n` thingie after all. I'll look it up."
- R looks for `n`, finding it in the Global Environment. Apparently it's bound to 2.
- R computes $2 + 3$ and returns 5. All is well.

Now call the function as follows:

```
addThree(n)
```

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

Again let's consider how this works:

- R sees the `n`. Since the function has only one parameter, R figures that you mean to assign the value of `n` (in some environment) to its parameter `n`.
- Everything now proceeds just as before, with 5 being the number returned.

Now let's remove `n` from the Global Environment:

```
rm(n)
```

Now call the function again, in the following way:

```
addThree(n = n)
```

```
Error in addThree(n = n) : object 'n' not found
```

Can you see why we got an error? This time when R goes looking for `n`, it can't find it: `n` is no longer in the Global Environment, nor is it anywhere else along the search path. Accordingly R throws the error.

The call `addThree(n)` will elicit the same error message, for the same reason.

The moral of the story is that parameters really, really are NOT the same thing as arguments, even when a parameter and an argument happen to be called by the same name.

3.5.3 Function Environments

Let's summarize what we have learned so far:

- An environment is a collection of names associated with objects.
- The Global Environment is the environment that is active when we are working from the console.
- When R needs to look up a name, it consults a search path.
- When we are in the Global Environment the search path starts there, and continues to:
  - the last package loaded (the parent environment),
  - the package before that (the "grandparent environment"),
  - and so on …
  - … up to package **base**.
- the first object of the right type having the given name that is found along the search path is the object to which R will associate the name.

Just as the Global Environment is a child of the last package loaded, so the Global Environment can have children of its own. In fact a child-environment is created whenever we define a function in the Global Environment and then run it.

Consider the following code:

```
a <- 10
b <- 4
f <- function(x, y) {
  a <- 5
  print(ls())
  cat("a is ", a, "\n",
      "b is ", b, "\n",
      "x is ", x, "\n",
      "y is ", y, "\n", sep = "")
}
```

Note that `a` and `b` are now in the Global Environment, where the value of `a` is 10 and the value of `b` is 5.

We have defined the function `f()`; pretty soon we will *call* it. The moment we do so, we will no longer be working directly from the console: instead R will hand control over to the function that it can execute the code in its body. This means that the Global Environment will no longer be the active environment. Instead the active environment will be one that is created at the moment when `f` is called. Accordingly, it is called the *run-time environment* (also known as the *evaluation environment*) of f.

Let's go ahead and call `f()`:

```
f(x = 2, y = 3)
```

```
## [1] "a" "x" "y"
## a is 5
## b is 4
## x is 2
## y is 3
```

In the body of the function `ls()` prints out all of the names in the active environment—which at the moment is the run-time environment of `f()`. This environment contains `a` with a value of 5—the `a` with a value of 10 is masked from it—along with the `x` and `y` that were passed into the function as arguments. The `a` variable having the value 5 that was created within the body of the function is said to be *local* to the function. Thus we can say that the run-time environment of a function consists of the variables that are local to the function and the arguments that were passed into it.

Observe that `b` is *not* a name in the function's run-time environment: instead it is in the Global Environment. Nevertheless R can "find" `b` from within the function because the R considers the Global Environment—the environment in which `f()` was defined—to be the parent of the run-time environment[6], and so the Global Environment is the second place R will look when searching for an object named `b`. Computer scientists say that `b` is *within the scope* of the function.

What happens to the run-time environment when `f()` finishes executing code? R simply destroys it. It's as if the `a`, `x` and `y` came to life "inside of" `f()` but died as soon as `f()` stopped working.

The next time `f()` is called, a new run-time environment will be created to enable the code in the body of `f()` to do its work.

One consequence of the ephemeral nature of run-time environments is that they are not accessible from parent environments. Thus if the active environment is the Global Environment and you run across a reference to `a`, you will never "find" the `a` "inside of" `f()` or "inside of" any other function, for that matter. R looks only in the active environment and in ancestor-environments, never in child-environments, and besides the run-time environment no longer exists after a function has been called.

Let's make sure of this with an example.

```
a <- 5
f <- function() {
  a <- 10
  print(ls.str()) # print out the active environment
}
f()
```

```
## a :  num 10
```

Did calling `f()` change the value of `a` in the Global Environment? Let's see:

```
a
```

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

Nope, `a` is still 10.

This is a very good thing. It would be very confusing if assignment to a variable within a function were to "change the values" of variables—happening to have the same name—that were declared outside of the function's environment.

[6]For any function that is created in R, the enclosing environment of the function is set to be the environment that was active when the function was defined. This feature is known as *lexical scoping*. Many other languages use *dynamic scoping*, meaning that the enclosing environment is the environment that is active when the function is *called*. At this stage in your work with R, when you almost always create functions while working the Global Environment, it can be a bit difficult to become aware of situations when the distinction between lexical and dynamic scoping makes a practical difference. However, the difference is there and it constantly affects your work with R, especially when you use a function from an R *package* (see Section 3.6 for more on packages). Since the environment associated with a package is the enclosing environment for any R-function defined in that package, functions from packages behave in a standard, expected way, no matter what environment—Global or otherwise—they are called in. For a practical application of lexical scoping that is not related to packages, consult Chapter 6 of (Grolemund (2014)).

3.6 A Note on Packages

We have seen that packages make up most of the search path when the active directory is the Global Environment. We have also mentioned a couple of packages explicitly—**mosaicData** and **ggplot2** back in Section 1.2.5 for example. But what exactly is a package?

A *package* is a bundle of R-code (usually functions) and data that is organized according to well-defined conventions and documented so that users can learn how to use the code and data. When someone bundles code into a package it becomes easy to share it with others and and to re-use it for one task after another.

3.6.1 Installed Packages

When you click on the Packages tab in the lower right-hand pane in R Studio, you can see a list of all the packages that are installed on the machine. You can get the same information by running the command:

```
installed.packages()[, c("Package", "Version")]
```

In fact R is really nothing but a collection of packages. Many of the R-functions you have been learning about come from the package **base**. This is one of a number of packages that are automatically attached to the search path when an R session begins. Other packages have to be attached by you if you want immediate access to all of the functions and data that they contain.

In order to attach a package, you can click the little box next to its name in the Package tab in R-Studio, or you can attach it from the console with the command:

```
library(<name of package here>)
```

When you don't want a package any more, you can detach it from the search path by un-clicking the little box, or by running this command:

```
detach("package:<name of package here>", unload=TRUE)
```

The package will still be installed, ready to be attached whenever you like.

3.6.2 Learning About a Package

You can learn about a Package by clicking on its name, or by using the command:

```
help(<name of package>)
```

From the display that shows in the Help pane you can navigate to learn about each of the functions and data sets that come with the package.

3.6.3 Installing Packages

You can also install additional packages on the computer. This can be done by clicking the Install button in R Studio and typing in the package name, or with the command:

```
install.packages("<name of package here>")
```

The package will be downloaded from the Comprehensive R Archive Network (CRAN) and installed in in your Home directory.

As long as we are working on the R Studio server, it's a good idea to refrain from installing packages yourself, unless they are packages that we don't use in class and that you simply want to explore on your own. That's because when you install your own packages on the Server they go into a special directory in your Home folder and become part of your "User Library". Packages that are installed by a system administrator for general use are in the "System Library." If a package is in your User Library and in the System Library, when you ask to attach it you will get the version that it is your User Library. Now packages are updated from time to time, so it may happen that the version you have in your User Library will be different from the one in the System Library. If that is the case then your package might not work the same way for you as it does for the instructor and for other students: that can be confusing.

Eventually, though, you will install R and R Studio on your own computer, and then you will have to install many packages on your own.

Not all packages come from CRAN: many useful packages exist on other repositories, including the very popular code repository known as GitHub[7]. Special functions exist to install R-packages from GitHub. For example, you may eventually wish to install the package **tigerData**, which resides in a GitHub repository belonging to your instructor. In order to install it, you would use the `install_github()` function from the **devtools** package (Pruim et al., 2016), like this:

```
devtools::install_github(repo = "homerhanumat/tigerData")
```

There are a couple of things worth noting about the command above:

1. The argument to `repo` has two parts: the word before the "/" is the username of the individual who owns the repository; the word after the "/" is the name of the repository itself. For R-packages on GitHub, the name of the repository is the same as the name of the package.
2. The double-colon `::` is used to access a function from a package, without having to attach the entire package. Thus `devtools::install_github()` refers to the function `install_github()` in package **devtools**. Similarly, if you want to access, say, just the `Births78` data set from the **mosaicData** package then you could refer to it as `mosaicData::Births78`.

3.7 More to Learn

By now we have have:

- learned about vectors, an important data structure in R;
- met quite a few R functions that help us manipulate vectors and print things to the console;
- learned how to write functions so that we can re-use solutions to problems.

But still it seems that—unless we can find a clever way to exploit vectorization—R isn't doing anything very impressive, really. In order to unlock the true powers of R (or any programming language, for that matter) we have to acquire more control over what expressions R will evaluate, how many times it will evaluate them, and under what conditions it will do so. This is the domain of *flow control*, the subject of our next chapter.

[7] `https://github.com/`

Glossary

Don't Repeat Yourself (DRY) A principle of computer programming that holds that general solutions should be set forth in one place and usable in many places, and that information needed in many places should be defined authoritatively in one place.

Parameters of a Function The parameters of a function (also called the *formal arguments* of the function) are the names that will be used in the body of the function to refer to the actual arguments supplied to the function when it is called.

Argument An argument for a function is a value that is assigned to one of the parameters of the function. (Sometimes arguments are called an *actual arguments* in order to distinguish then from parameters that are often called *formal arguments*.)

> "It's useful to distinguish between the formal arguments and the actual arguments of a function. The formal arguments are a property of the function, whereas the actual or calling arguments can vary each time you call the function."
>
> —H. Wickham, *Advanced R Programming*

Body of a Function The body of a function is the code that is executed when the function is called. In R, when the body consists of more than one expression then it appears inside of curly braces.

Side-Effect Any result produced outside of the run-time environment of a function, other than the value that the function returns.

Default Value A value for a parameter of a function that is provided when the function is defined. This value will become the value assigned to the parameter when the function is called, unless the user explicitly assigns some other value as the argument.

Environment An object stored in the computer's memory that keeps track of name-value pairs.

Active Environment The environment that R will consult first in order to find the value of any name in an expression.

Global Environment The environment that is active when one is using R from the console.

Parent Environment The second environment (after the active environment) that R will search when it needs to look up a name.

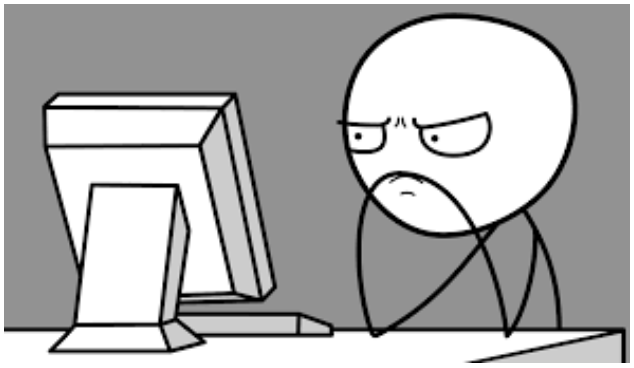
Run-time Environment (also called the "Evaluation Environment") A special environment that is created when a fuction is called and ceases to exist when the function finishes executing. It contains the values that are local to the function and the arguments of the function as well.

Scoping The process by which the computer looks up the object associated with a name in an expression.

Search Path The sequence of environments that the computer will consult in order to find an object associated with a name in an expression. The sequence begins with the active environment, followed by its parent environment, followed by the parent of the parent environment, and so on.

Package A bundle of R-code and data that is organized according to well-defined conventions and documented so that users can learn how to use the code and data.

Exercises



1. Write a function called `pattern()` that when given a character will will print out the character in a pattern like this:

   ```
   *
   **
   ***
   **
   *
   ```

   That is: a row of one, then a row of two, then a row of three, then a row of two, and finally a row of one.

   The function should take one parameter called `char`. The default value of this parameter should be `*`. Typical examples of use should be as follows:

   ```
   pattern()
   ```

   ```
   ## *
   ## **
   ## ***
   ## **
   ## *
   ```

   ```
   pattern(char = "x")
   ```

   ```
   ## x
   ## xx
   ## xxx
   ## xx
   ## x
   ```

2. Write a function called `reverse()` that, given any vector, returns a vector with the elements in reverse order. It should take one parameter called `vec`. The default value of `vec` should be the vector `c("Bob", "Marley")`. Typical examples of use should be:

   ```
   reverse()
   ```

   ```
   ## [1] "Marley" "Bob"
   ```

```
reverse(c(3,2,7,6))
```

```
## [1] 6 7 2 3
```

Hint: Recall how you can use sub-setting to reverse:

```
firstFiveLetters <- c("a", "b", "c", "d", "e")
firstFiveLetters[5:1]
```

```
## [1] "e" "d" "c" "b" "a"
```

You just need to figure out how to reverse vectors of arbitrary length.

3. A vector is said to be a *palindrome* if reversing its elements yields the same vector. Thus, `c(3,1,3)` is a palindrome, but `c(3,1,4)` is not a palindrome.

   Write a function called `isPalindrome()` that, when given any vector, will return `TRUE` if the vector is a palindrome and `FALSE` if it is not a palindrome. The function should take a single parameter called `vec`, with no default value. Typical examples of use should be:

```
isPalindrome(vec = c("Bob", "Marley", "Bob"))
```

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

```
isPalindrome(c(3,2,7,4,3))
```

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

   Hint: You already have the function `reverse()` from the previous Exercise. Use this function, along with the Boolean operator `==` and the `all()` function.

4. The eighteenth-century mathematician Leonhard Euler discovered that:

$$\frac{\pi^2}{6} = \sum_{k=1}^{k=\infty} \frac{1}{k^2}.$$

   It follows that

$$\pi = \sqrt{\left(\sum_{k=1}^{k=\infty} \frac{6}{k^2}\right)}.$$

   Use this fact to write a function called `eulerPI()` that will approximate π. The function should take a single parameter `n`, which is the number of terms in the infinite series that are to be summed to make the approximation. The default value of `n` should be 10,000.

Chapter 4

Flow-Control

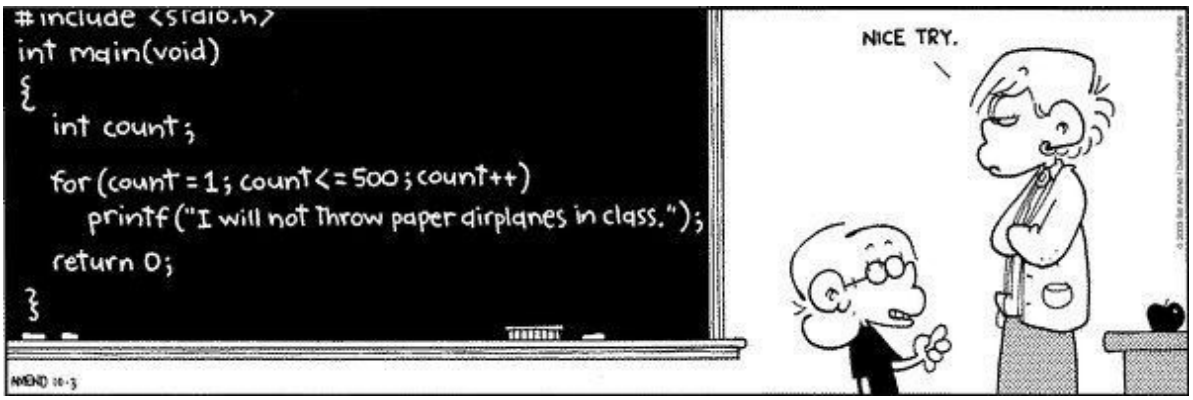


Figure 4.1: Foxtrot, October 3, 2003. Used with permission.

In the above scene from the comic-strip *Foxtrot*, young Jason has attempted to short-cut his "write-on-the-blackboard" punishment via some code in the C-language that would print out his assigned sentence 500 times. This is an example of *flow control*.

Flow control encompasses the tools in a programming language that allow the computer to make decisions and to repeat tasks. In this Chapter we will learn how flow control is implemented in R.

4.1 Prologue: Prompting the User

Before we get started with flow control itself, let's address the question of how we might extract information from someone who runs our our code.

Up to this point any information that we have wanted to process has had to be entered into the code itself. For example, if we want to print out a name to the console, we have to provide that name in the code, like this:

```
person <- "Dorothy"
cat("Hello, ", person, "!\n", sep = "")
```

```
## Hello, Dorothy!
```

We can get any printout we like as long as we assign the desired string to `person`—but we have to do that *in the code itself*. But what if we don't know the name of the person whom we want to greet? How can we be assured of printing out the correct name?

The `readline()` function will be helpful, here. It reads input directly from the console and produces a character vector (of length one) that we may use as we like. Thus

```
person <- readline(prompt = "What is your name?  ")
# Type your name before running the next line!
cat("Hello, ", person, "!\n", sep = "")
```

```
## What is your name?  Cowardly Lion
## Hello, Cowardly Lion!
```

Note that the input obtained from `readline()` is always a character vector of length one—a single string—so if you plan to use it as a number then you need to convert it to a number. The `as.numeric()` function will do this for you:

```
number <- readline("Give me a number, and I'll add 10 to it:  ")
# Give R your number before you run the next line!
cat("The sum is:  ", as.numeric(number) + 10)
```

```
## Give me a number, and I'll add 10 to it:  15
## The sum is:  25
```

4.2 Making Decisions: Conditionals

We are now ready to begin addressing flow control itself. We'll begin by looking at the various facilities R has for making decisions.

4.2.1 If Statements

Let's design a simple guessing-game for the user:

- The computer will pick randomly a whole number between 1 and 4.
- The user will then be asked to guess the number.
- If the user is correct, then the computer will congratulate the user.

```
number <- sample(1:4, size = 1)
guess <- as.numeric(readline("Guess the number (1-4):  "))
if ( guess == number ) {
  cat("Congratulations!  You are correct.")
}
```

The `sample()` function randomly picks a value from the vector that is is given. The `size` parameter specifies how many numbers to pick. (This time we only want one number.)

Flow control enters the picture with the reserved word `if`. Immediately after `if` is a Boolean expression enclosed in parentheses. This expression is often called the *condition*. If the condition evaluates to `TRUE`, then the body of the `if` statement—the code enclosed in the brackets—will be executed. On the other hand if the condition evaluates to `FALSE`, then R skips past the bracketed code.[1]

The general form of an `if` expression is as follows:

```
if ( condition ) {
  ## code to run when the condition evaluates to TRUE
}
```

The code above congratulates the a lucky guesser, but it has nothing at all to say to someone who did not guess correctly. The way to provide an alternative is through the addition of the `else` reserved-word:

```
number <- sample(1:4, size = 1)
guess <- as.numeric(readline("Guess the number (1-4):  "))
if ( guess == number ) {
  cat("Congratulations!  You are correct.")
} else {
  cat("Sorry, the correct number was ", number, ".\n", sep = "")
  cat("Better luck next time!")
}
```

The general form of an `if-else` expression is as follows:

```
if ( condition ) {
  # code to run if the condition evaluates to TRUE
} else {
  # code to run if condition evaluates to FALSE
}
```

An `if-else`can be followed by any number of `if-else`'s, setting up a chain of alternative responses:

```
number <- sample(1:4, size = 1)
guess <- as.numeric(readline("Guess the number (1-4):  "))
if ( guess == number ) {
  cat("Congratulations!  You are correct.")
} else if ( abs(guess - number) == 1 ){
  cat("You were close!\n")
  cat("The correct number was ", number, ".\n", sep = "")
```

[1] Actually you don't need the brackets if you plan to put only one expression in them. Many people keep the brackets, though, for the sake of clarity.

```
} else {
  cat("You were way off.\n")
  cat("The correct number was ", number, ".\n", sep = "")
}
```

In general, a chain looks like this:

```
if ( condition1) {
  # code to run if condition1 evaluates to TRUE
} else if ( condition2 ) {
  # code to run if condition2 evaluates to TRUE
} else if ( condition3 ) {
  # code to run if condition2 evaluates to TRUE
} else if ......

  # and so on until
} else if ( conditionN ) {
  # code to run if conditionN evaluates to TRUE
}
```

4.2.2 Application: Invisible Returns

Let's think again about the π-computing function from Section 3.4.1:

```
madhavaPI <- function(n = 1000000) {
  k <- 1:n
  terms <- (-1)^(k+1)*4/(2*k-1)
  sum(terms)
}
```

We could use `if` to write in a "talky" option:

```
madhavaPI <- function(n = 1000000, verbose = FALSE) {
  k <- 1:n
  terms <- (-1)^(k+1)*4/(2*k-1)
  approx <- sum(terms)
  if ( verbose) {
    cat("Madhava's approximation is:  ", approx, ".\n", sep = "")
    cat("This is based on ", n, " terms.\n", sep = "")
  }
  approx
}
```

Try it out:

```
madhavaPI(n = 1000, verbose = TRUE)
```

```
## Madhava's approximation is:  3.140593.
## This is based on 1000 terms.
```

```
## [1] 3.140593
```

It's a bit awkward that the approximation gets printed out at the end: after the message on the console, the user doesn't need to see it. But if we were to remove the final `approx` expression, then the function would not return an approximation that could be used for further computations.

The solution to this dilemma is R's `invisible()` function.

```
madhavaPI <- function(n = 1000000, verbose = FALSE) {
  k <- 1:n
  terms <- (-1)^(k+1)*4/(2*k-1)
  approx <- sum(terms)
  if ( verbose) {
    cat("Madhava's approximation is:  ", approx, ".\n", sep = "")
    cat("This is based on ", n, " terms.\n", sep = "")
  }
  invisible(approx)
}
```

If you wrap an expression in `invisible()`, then it won't be printed out to the console:

```
madhavaPI(n = 1000, verbose = TRUE)
```

```
## Madhava's approximation is:  3.140593.
## This is based on 1000 terms.
```

Nevertheless it is still returned, as we can see from the following code, in which the approximation is computed without any output to the console and stored in the variable `p` for use later on in a `cat()` statement.

```
p <- madhavaPI() # verbose is FALSE by default
cat("Pi plus 10 is about ", p + 10, ".", sep = "")
```

```
## Pi plus 10 is about 13.14159.
```

4.2.3 Ifelse

The `ifelse()` function is a special form of the if-else construct that is used to make assignments, and is especially handy in the context of vectorization.

Suppose that you have a lot of heights:

```
height <- c(69, 67, 70, 72, 65, 63, 75, 70)
```

You would like to classify each person as either "tall" or "short", depending on whether they are respectively more or less than 71 inches in height. `ifelse()` makes quick work of it:

```
heightClass <- ifelse(test = height > 70,
                      yes = "tall", no = "short")
heightClass
```

```
## [1] "short" "short" "short" "tall"  "short" "short" "tall"  "short"
```

Note that `ifelse()` takes three parameters:

- `test`: the condition you want to evaluate;
- `yes`: the value that gets assigned when `test` is true;
- `no`: the value assigned when `test` is false;

Most programmers don't name the parameters. This is fine—just remember to keep the test-yes-no order:

```
ifelse(height > 70, "tall", "short")
```

```
## [1] "short" "short" "short" "tall"  "short" "short" "tall"  "short"
```

Here's another example of the power of `ifelse()`. If a triangle has three sides of length x, y and z, then the sum of any two sides must be greater than the remaining side:

$$x + y > z,$$
$$x + z > y,$$
$$y + z > x.$$

This fact is known as the *Triangle Inequality*. It works the other way around, too: if three positive numbers are such that the sum of any two exceeds the third, then three line segments having those numbers as lengths could be arranged into a triangle.

We can write a function that, when given three lengths, determines whether or not they can make a triangle:

```
isTriangle <- function(x, y, z) {
  (x + y > z) & (x +z > y) & (y + z > x)
}
```

`isTriangle()` simply evaluates a Boolean expression involving `x`, `y` and `z`. It will return `TRUE` when the three quantities satisfy the Triangle Inequality; otherwise, it returns `FALSE`. Let's try it out:

```
isTriangle(x = 3, y = 4, z = 5)
```

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

Recall that Boolean expressions can involve vectors of any length. So suppose that we are would like to know which of the following six triples of numbers could be the side-lengths of a triangle:

$$(2, 4, 5), (4.7, 1, 3.8), (5.2, 8, 12), (6, 6, 13), (6, 6, 11), (9, 3.5, 6.2)$$

We could enter the triples one at a time into `isTriangle()`. On the other hand we could arrange the sides into three vectors of length six each:

```
a <- c(2, 4.7, 5.2, 6, 6, 9)
b <- c(4, 1, 2.8, 6, 6, 3.5)
c <- c(5, 3.8, 12, 13, 11, 6.2)
```

Then we can decide about all six triples at once:

```
isTriangle(x = a, y = b, z = c)
```

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

We could also use `ifelse()` to create a new character-vector that expresses our results verbally:

```
triangle <- ifelse(isTriangle(a, b, c), "triangle", "not")
triangle
```

```
## [1] "triangle" "triangle" "not"      "not"      "triangle" "triangle"
```

4.2.4 Switch

If you have to make a decision involving two or more alternatives you can use a chain of `if ... else` constructions. When the alternatives involve no more than the assignment of a value to a variable, you might also consider using the `switch()` function.

For example, suppose that you have days of the week expressed as numbers. Maybe it's like this:

- 1 stands for Sunday
- 2 for Monday
- 3 for Wednesday
- and so on.

If you would like to convert a day-number to the right day name, then you could write a function like this:

```
dayWord <- function(dayNumber) {
  switch(dayNumber,
         "Sunday",
         "Monday",
         "Tuesday",
         "Wednesday",
         "Thursday",
         "Friday",
         "Saturday")
}
dayWord(3)
```

```
## [1] "Tuesday"
```

In `switch()` above, the first argument after `dayNumber` is what goes with 1, the second argument is what goes with 2, and so on.

When the item you want to convert is a string rather than a number, then the `switch()` function works a little bit differently. Suppose, for instance, that you want to *abbreviate* the names of the weekdays. You might write an abbreviation-function as follows:

```
abbrDay <- function(day) {
  switch(day,
         Monday = "Mon",
         Tuesday = "Tue",
         Wednesday = "Wed",
         Thursday = "Th",
         Friday = "Fri",
         Saturday = "Sat")
}
abbrDay("Wednesday")
```

```
## [1] "Wed"
```

In the above call to `switch()`, the weekday names you want to abbreviate appear as the names of named character-vectors, each of length one. The value of each vector is what the its name will be converted to.

When you are converting strings you have the option to provide a default conversion for values that don't fit into the pattern you have in mind. All you have to do is to provide the default value as an additional argument. (It should NOT have a name.) Thus:

```
abbrDay <- function(day) {
  switch(day,
         Monday = "Mon",
         Tuesday = "Tue",
         Wednesday = "Wed",
         Thursday = "Th",
         Friday = "Fri",
         Saturday = "Sat",
         "not a weekday!")
}
abbrDay("Wednesday")
```

```
## [1] "Wed"
```

```
abbrDay("Neptune")
```

```
## [1] "not a weekday!"
```

4.3 Repeating Things: Looping

We have looked a bit into the aspect of flow control that pertains to making decisions. Let us now turn to the R-constructs that make the computer repeat actions.

4.3.1 For Loops

The reserved word `for` is used to make R repeat an action a specified number of times.

We begin with an very simple example:

```
for ( i in 1:4 ) {
  cat("Hello, Dorothy!\n")
}
```

```
## Hello, Dorothy!
## Hello, Dorothy!
## Hello, Dorothy!
## Hello, Dorothy!
```

Here's how the loop works. Recall that the vector `1:4` is simply the sequence of integers from 1 to 4:

```
1:4
```

```
## [1] 1 2 3 4
```

When R sees the code `(i in 1:4)` it knows that it will have to go four times through the body of the loop.

```
cat("Hello, Dorothy!\n")
```

(The *body* of a loop is what's contained in the brackets after `for(i in 1:4)`). At the start of the loop, the *index variable* `i` is set to 1. After the body is executed the first time, R sets `i` to 2, then executes the body again. Then R sets `i` to 3 and executes the body yet again. Then R sets `i` to 4, and executes the body for the final time. The result is four lines printed out to the console.

The more you need to repeat a particular pattern, the more it makes sense to write your code with a loop.

The general form of a loop is:

```
for ( var in seq ) {
  # code involving var
}
```

`var` is the index variable, and it can be any permitted name for a variable, and `seq` can be any vector. As R traverses the loop, the value of the index variable `var` becomes each element of the vector `seq` in turn. With every change in the value of `var`, the code in the brackets is executed.

To *iterate* is to do a thing again and again. The vector `seq` is sometimes called an *iterable*, since it is "iterated over." It contains the values that the index variable will assume, one by one, as the loop is repeated.

It's important to realize that the index variable can have *any* valid name, and the sequence can be any type of vector at all—not just a sequence of consecutive whole numbers beginning with 1. This level of generality permits a for-loop to automate a wide variety of tasks, and for its code to be written in a way that evokes the operations being performed.

For example, here is a loop to print out some greetings;

```
people <- c("Dorothy", "Tin Man", "Scarecrow", "Lion")
for ( person in people ) {
  cat("Hello, ", person, "!\n", sep = "")
}
```

```
## Hello, Dorothy!
## Hello, Tin Man!
## Hello, Scarecrow!
## Hello, Lion!
```

Or perhaps you want to abbreviate a vector-ful of weekday-names:

```
weekdays <- c("Saturday", "Monday", "Friday", "Saturday")
for ( day in weekdays ) {
  print(abbrDay(day))
}
```

```
## [1] "Sat"
## [1] "Mon"
## [1] "Fri"
## [1] "Sat"
```

Quite often you will want to store the results of your trips through the loop. Let's do this for our abbreviated weekdays:

```
weekdays <- c("Saturday", "Monday", "Friday", "Saturday")
abbrDays <- character(length(weekdays))
```

We used the `character()` function to create the character vector `abbrDays`. We specified that the number of elements in `abbrDays` shall be the same as the number of elements in `weekdays`. Right now `abbrDays` isn't very interesting, as all of its elements are empty strings:

```
abbrDays
```

```
## [1] "" "" "" ""
```

We will now write a for-loop to fill it up with abbreviations:

```
for ( i in 1:length(weekdays) ) {
  abbrDays[i] <- abbrDay(weekdays[i])
}
```

Now each of the four elements of `abbrDays` contains the abbreviation for the corresponding element of `weekdays`:

```
abbrDays
```

```
## [1] "Sat" "Mon" "Fri" "Sat"
```

You will often have reason to set up an empty vector of a definite length and use a loop to store information in it. The general format looks like this:

```
results <- numeric(someLength) # empty numerical vector

for ( i in 1:someLength ) {
  # computations involving i, and then:
  results[i] <- some_result_depending_on_i
}

# then do something with results, such as:
print(results)
```

4.3.2 For-Loop Caution

You might need to exercise some caution in your choice name for the index variable. If you are already using it as the name for a variable in the same environment, then you will overwrite that variable, as demonstrated by the following code:

```
day <- "Thursday"
cat("Today is ", day, ".\n", sep = "")
```

```
## Today is Thursday.
```

```
weekdays <- c("Saturday", "Monday", "Friday", "Saturday")
abbrDays <- character(length(weekdays))
for ( day in weekdays ) {
  print(day)
}
```

```
## [1] "Saturday"
## [1] "Monday"
## [1] "Friday"
## [1] "Saturday"
```

```
cat("Today is ", day, ".\n", sep = "")
```

```
## Today is Saturday.
```

Of course, that won't happen if the for-loop is inside of a function and your variable is outside of it.

```
day <- "Thursday"
cat("Today is ", day, ".\n", sep = "")
```

```
## Today is Thursday.
```

```
weekdays <- c("Saturday", "Monday", "Friday", "Saturday")
listDays <- function(days) {
  for ( day in days )    {
    print(day)
  }
}
listDays(weekdays)
```

```
## [1] "Saturday"
## [1] "Monday"
## [1] "Friday"
## [1] "Saturday"
```

```
cat("Today is still ", day, ".\n", sep = "")
```

```
## Today is still Thursday.
```

4.3.3 Breaking Out of a Loop

Sometimes you finish the task at hand before you are done with the loop. If this is a possibility for you then you may arrange to break out of the loop with the `break` reserved-word.

Suppose for example that you want a function that searches through a vector for a given element, updating the user along the way as to the progress of the search. You can try something like this:

```
# function to find index of element in vector.
# returns -1 if elem is not in vector
verboseSearch <- function(elem, vec) {
```

```
  # The following logical keeps track of whether
  # we have found the element.
  # We have not yet begun the search so start it
  # at FALSE.
  found <- FALSE

  # validate input:
  if ( length(vec) == 0 ) {
    cat("The vector empty.  No way does it contain ",
        elem, ".", sep = "")
    return(-1)
  }

  # check the elements of vector:
  for ( i in 1: length(vec) ) {
    if ( vec[i] == elem ) {
      # record that we found the element:
      found <- TRUE
      break
    } else {
      # report no match at this index:
      cat("Checked index ", i,
          " in the vector.  No match there ...\n", sep = "")
    }
  }

  if ( found ) {
    # report success:
    cat("Found ", elem, " at index ", i, ".\n", sep = "")
    return(i)
  } else {
    # report failure:
    cat(elem, " is not in the vector.\n", sep = "")
    return(-1)
  }
}
```

Let's see our function in action:

```
people <- c("Dorothy", "Tin Man", "Scarecrow", "Lion")
scarecrowPlace <- verboseSearch("Scarecrow", people)
```

```
## Checked index 1 in the vector.  No match there ...
## Checked index 2 in the vector.  No match there ...
## Found Scarecrow at index 3.
```

In the code for `verboseSearch()` you will notice that there is an initial check on the length of the vector. This is actually important. If a user were to enter an empty vector, then it length would be 0. Then in the loop the sequence would be `1:0`, which is the vector with elements 1 and 0. But look at what happens when you ask for any element of a zero-length vector:

```
emptyVec <- character(0)
emptyVec[1]  # You get NA
```

```
## [1] NA
```

Then check out what happens if you compare an `NA` to a string:

```
NA == "Scarecrow"
```

```
## [1] NA
```

Now look at what happens in an `if` statement where the condition is `NA`:

```
if ( NA ) {
  cat("We are in the bracket!\n")
}
```

```
## Error in if (NA) { : missing value where TRUE/FALSE needed
```

Checking that the input vector has positive length is an example of *validating* input. When you write functions for other people to use you will find that it's important to validate their input instead of allowing R to throw obscure error messages at them.

4.3.4 Solving the Empty-Vector Problem in `for`-Loops with `seq_along()`

In the previous section we considered a possible problem with `for`-loops of the following form:

```
for ( i in 1:length(vec) ) {
  ## do something ...
}
```

In the above loop `vec` could be thought of as a "loop-defining" vector: its length determines the sequence of values 1, 2, 3 … for the index `i`. This sequence of values is supposed to end at the length of `vec`.

The problem is that if `vec` happens to be an empty vector then we probably don't want to enter the loop at all. However, the length of an empty vector is 0, and so the vector `1:length(vec)` actually works out to be a vector with two elements:

```
c(1,0)
```

Hence R will go through the loop twice: once when `i` is 1, and again when `i` is 0. Depending on what the loop does, very unexpected results could be produced.

In the previous section we dealt with the problem by writing an `if`-statement that provides the proper response when `vec` is empty. In many circumstances however, all we need to do is to make sure that the loop is skipped when the vector is empty.

A handy way to ensure skipping is to use the function `seq_along()` . Given any non-empty vector, `seq_along()` produces a sequence-vector that begins with 1 and ends with the length of the vector, thus:

```
vec <- c("a", "d", "f")
seq_along(vec)
```

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

On the other hand, if the vector is empty, then `seq_long()` returns an empty numeric vector:

```
vec <- character()
seq_along(vec)
```

```
## integer(0)
```

Now consider the loop inside the following function:

```
loopy <- function(vec) {
  for ( i in seq_along(vec) ) {
    cat("This time i is ", i, ".\n", sep = "")
  }
}
```

Given a non-empty vector, it goes through the loop a number of times equal to the length of the vector:

```
loopy(c("a", "d", "f"))
```

```
## This time i is 1.
## This time i is 2.
## This time i is 3.
```

On the other hand, when given an empty vector the function does not enter the loop at all:

```
loopy(character())  # no output to console!
```

When you are writing a program that is complex enough that you don't know whether the loop-defining vector might be empty, it is good practice to use `seq_along()` as a safeguard.

4.3.5 Skipping Ahead in a Loop

Depending on what happens within a loop, you might sometimes wish to skip the remaining code within the loop and proceed to the next iteration. R provides the reserved-word `next` for this purpose. Here is a simple example:

```
vec <- c("a","e", "e", "i", "o", "u", "e", "z")
# shout ahoy when you see the specified element
verboseAhoy <- function(elem, vec) {
  if (length(vec) > 0) {
    for ( i in 1: length(vec) ) {
      if ( vec[i] != elem) next
      cat("Ahoy! ", elem, " at index ", i, "!\n", sep = "")
    }
  }

}
verboseAhoy("e", vec)
```

```
## Ahoy! e at index 2!
## Ahoy! e at index 3!
## Ahoy! e at index 7!
```

When the `vec[i] == elem` condition is true, R immediately skips the rest of the loop, increments the value of the index variable `i`, and runs through the loop again.

You can always accomplish the skipping without using `next` explicitly, but it's nice to have on hand.

4.3.6 Repeat

For-loops are pretty wonderful, but they are best used in circumstances when you know how many times you will need to loop. When you need to repeat a block of code until a certain condition occurs, then the `repeat` reserved-word might be a good choice.

For example, suppose you want to play the number-guessing game with the user, but let her keep guessing until either she gives up or gets the correct answer. Here's an implementation using `repeat`:

```
n <- 20
number <- sample(1:n, size = 1)
cat("I'm thinking of a whole number from 1 to ", n, ".\n", sep = "")
repeat {
  guess <- readline("What's your guess? (Enter q to quit.)  ")
  if ( guess == "q"  ) {
    cat("Bye!\n")
    break
  } else if ( as.numeric(guess) == number ) {
    cat("You are correct!  Thanks for playing!")
    break
  }
  # If we get here, the guess was not correct:
  # loop will repeat!
}
```

The game works well enough, but if you give it a try, you are sure to find it a bit fatiguing. It wold be nice to give the user a hint after an incorrect guess. Let's revise the game to tell the reader whether her guess was high or low. While we are at it, let's cast the game into the form of a function.

```
numberGuess <- function(n) {
  number <- sample(1:n, size = 1)
  cat("I'm thinking of a whole number from 1 to ", n, ".\n", sep = "")
  repeat {
    guess <- readline("What's your guess? (Enter q to quit.)  ")
    if (guess == "q") {
      cat("Bye!\n")
      break
    } else if (as.numeric(guess) == number) {
      cat("You are correct!  Thanks for playing!")
      break
    }
    # If we get to this point the guess was not correct.
    # Issue hint:
    hint <- ifelse(as.numeric(guess) > number, "high", "low")
```

```
    cat("Your guess was ", hint, ".  Keep at it!\n", sep = "")
    # Repeat loop
  }
}
```

A typical game:

```
> numberGuess(100)
I'm thinking of a whole number from 1 to 100.
What's your guess? (Enter q to quit.)  50
Your guess was high.  Keep at it!
What's your guess? (Enter q to quit.)  25
Your guess was high.  Keep at it!
What's your guess? (Enter q to quit.)  12
Your guess was low.  Keep at it!
What's your guess? (Enter q to quit.)  18
Your guess was low.  Keep at it!
What's your guess? (Enter q to quit.)  22
Your guess was high.  Keep at it!
What's your guess? (Enter q to quit.)  20
You are correct!  Thanks for playing!
```

4.3.7 While

The reserved word `while` constructs a loop that runs as long as a specified condition is true. Unlike `repeat`, which launches directly into the loop, the condition for `while` is evaluated prior to the body of the loop. If the condition is false at the beginning, the code in the body of the loop is never executed.

`verboseSearch()` could be re-written with `while`:

```
verboseSearch <- function(elem, vec) {
  found <- FALSE
  if ( length(vec) == 0 ) {
    cat("The vector is empty.  No way does it contain ",
        elem, ".\n", sep = "")
    return(-1)
  }
  # index of vec (start looking at 1):
  i <- 1
  while ( !found & i <= length(vec) ) {
    if ( vec[i] == elem ) {
      found <- TRUE
      break
    }
    cat("No match at position ", i, " ...\n")
    i <- i + 1
  }

  if ( found ) {
    # report success:
    cat("Found ", elem, " at index ", i, ".\n", sep = "")
    return(i)
```

```
  } else {
    # report failure:
    cat(elem, " is not in the vector.\n", sep = "")
    return(-1)
  }
}
```

4.4 Application: The Collatz Conjecture

Take any positive integer greater than 1. Apply the following rule, which we will call the Collatz Rule:

- If the integer is even, divide it by 2;
- if the integer is odd, multiply it by 3 and add 1.

Now apply the rule to the resulting number, then apply the rule again to the number you get from that, and so on.

For example, start with 13. We proceed as follows:

- 13 is odd, so compute $3 \times 13 + 1 = 40$.
- 40 is even, so compute $40/2 = 20$.
- 20 is even, so compute $20/2 = 10$.
- 10 is even, so compute $10/2 = 5$.
- 5 is odd, so compute $3 \times 5 + 1 = 16$
- 16 is even, so compute $16/2 = 8$.
- 8 is even, so compute $8/2 = 4$.
- 4 is even, so compute $4/2 = 2$.
- 2 is even, so compute $2/2 = 1$.
- 1 is odd, so compute $3 \times 1 + 1 = 4$.
- 4 is even, so compute $4/2 = 2$.
- 2 is even, so compute $2/2 = 1$.

If we keep going, then we will cycle forever:

$$4, 2, 1, 4, 2, 1, \ldots$$

In mathematics the *Collatz Conjecture* is the conjecture that for *any* initial positive number, every Collatz Sequence (the sequence formed by repeated application of the Collatz Rule) eventually contains a 1, after which it must cycle forever. No matter how large a number we begin with, we have always found that it returns to 1, but mathematicians do not know if this will be so for *any* initial number.

A sequence of Collatz numbers can bounce around quite remarkably before descending to 1. Our goal in this section is to write a function called `collatz()` that will compute the Collatz sequence for any given initial number and draw a graph of the sequence as well.

First, let's make a function just for the Collatz Rule itself:

```
collatzRule <- function(m) {
  if ( m %% 2 == 0) {
    return(m/2)
  } else {
    return(3*m + 1)
  }
}
```

(Recall that `m %% 2` is the remainder of `m` after it is divided by 2. If this is 0, then `m` is even.)

Next let's try to get a function going that will print out Collatz numbers:

```
collatz <- function(n) {
  # n is the initial number
  while ( n > 1 ) {
    cat(n, " ", sep = "")
    # get the next number and call it n:
    n <- collatzRule(n)
  }
}
```

Let's try it out:

```
collatz(13)
```

```
## 13 40 20 10 5 16 8 4 2
```

So far, so good, but if we are going to graph the numbers, then we should store them in a vector. The problem is that we don't know how long the vector needs to be.

One possible solution is to add to the vector as we go, like this:

```
collatz <- function(n) {
  numbers <- numeric()
  while ( n > 1 ) {
    # stick n onto the end of numbers:
    numbers <- c(numbers, n)
    n <- collatzRule(n)
  }
  print(numbers)
}
```

Try it out:

```
collatz(13)
```

```
## [1] 13 40 20 10  5 16  8  4  2
```

This looks good. There are two problems, though, if the Collatz sequence happens to go on for a very long time.

- **Computation Time**: the user doesn't know when the sequence will end—if ever!—so she won't know whether a delay in production of the output is due to a long sequence or a problem with the program itself. as the sequence gets longer, the computation-time is made even longer by the way the following line of code works:

  `numbers <- c(numbers, n)`

  R cannot actually "stick" a new element onto the end of a vector. What it actually does is to move to a new place in memory and create an entirely new vector consisting of all the elements of `numbers` followed by the number `n`. R then assigns the name `numbers` to this value, freeing up the old place in memory where the previous `numbers` vector lived, but when a vector is very long copying can take a long time.

- **Memory Problems** Once the `numbers` vector gets long enough it will use all of the memory in the computer that is available to R. R will crash.

In order to get around this problem, we should impose a limit on the number of Collatz numbers that we will compute. We'll set the limit at 10,000. The user can change the limit, but should exercise caution in doing so. Also, we'll initialize our `numbers` vector to have a length set to this limit. We can then assign values to elements of an *already existing* vector: this is much faster than copying entire vectors from scratch.

```
collatz <- function(n, limit = 10000) {
  # collatz numbers will go in this vector
  numbers <- numeric(limit)
  # keep count of how many numbers we have made:
  counter <- 0
  while ( n > 1 & counter < limit) {
    # need to make a new number
    counter <- counter + 1
    # put the current number into the vector
    numbers[counter] <- n
    # make next Collatz number
    n <- collatzRule(n)
  }
  # find how many Collatz numbers we made:
  howMany <- min(counter, limit)
  # print them out:
  print(numbers[1:howMany])
}
```

Again let's try it:

```
collatz(257)
```

```
##   [1]  257  772  386  193  580  290  145  436  218  109  328  164   82   41
##  [15]  124   62   31   94   47  142   71  214  107  322  161  484  242  121
##  [29]  364  182   91  274  137  412  206  103  310  155  466  233  700  350
##  [43]  175  526  263  790  395 1186  593 1780  890  445 1336  668  334  167
##  [57]  502  251  754  377 1132  566  283  850  425 1276  638  319  958  479
##  [71] 1438  719 2158 1079 3238 1619 4858 2429 7288 3644 1822  911 2734 1367
##  [85] 4102 2051 6154 3077 9232 4616 2308 1154  577 1732  866  433 1300  650
##  [99]  325  976  488  244  122   61  184   92   46   23   70   35  106   53
## [113]  160   80   40   20   10    5   16    8    4    2
```

Things are working pretty well, but since the sequence of numbers might get pretty long, perhaps we should only print out the length of the sequence, and leave it to the reader to say whether the sequence itself should be shown.

```
collatz <- function(n, limit = 10000) {
  numbers <- numeric(limit)
  counter <- 0
  while ( n > 1 & counter < limit) {
    counter <- counter + 1
    numbers[counter] <- n
    n <- collatzRule(n)
  }
```

```
  howMany <- min(counter, limit)
  cat("The Collatz sequence has ", howMany, " elements.\n", sep = "")
  show <- readline("Do you want to see it (y/n)?  ")
  if ( show == "y" ) {
    print(numbers[1:howMany])
  }
}
```

Next let's think about the plot. We'll use the plotting system in the **ggplot2** package (Wickham and Chang, 2017).

```
library(ggplot2)
```

Later on we will make a serious study of plotting with **ggplot2**, but for now let's just get the basic idea of plotting a set of points. First, let's get a small set of points to plot:

```
xvals <- c(1, 2, 3, 4, 5)
yvals <- c(3, 7, 2, 4, 3)
```

`xvals` contains the x-coordinates of our points, and `yvals` contains the corresponding y-coordinates.

We set up a plot as follows:

```
p <- ggplot(mapping = aes(x = xvals, y = yvals))
```

The `ggplot()` function sets up a basic two-dimensional grid. The `mapping` parameter explains how data will be "mapped" to particular positions on the plot. In this case it has been set to:

`aes(x = xvals, y = yvals)`

`aes` is short for "aesthetics", which has to do with how somethings *looks*. The expression means that `xvals` will be located on the x-axis and `yvals` will be located on the y-axis of the plot.

Note that the entire plot has been assigned to the variable `p`. If we want to see the plot we could print p out (see Figure 4.2):

```
print(p)
```

The plot is blank! Why is this? Well, although `ggplot()` has been told what values are to be represented on the plot and where they might go, it has not yet been told how they should be shaped: it has not been told their *geometry*, you might say. We can add a geometry to `p` to get a picture (see Figure 4.3:

```
print(p + geom_point())
```

The geometry determines a lot about the look of the plot. In order to have the points connected by lines we could add `geom_line()` (see Figure 4.4:

```
print(p + geom_point() + geom_line())
```

We'll choose a scatterplot with connecting lines for our graph of the sequence. With a little more work we can get nice labels for the x and y-axes, and a title for the graph. Our `collatz()` function now looks like:

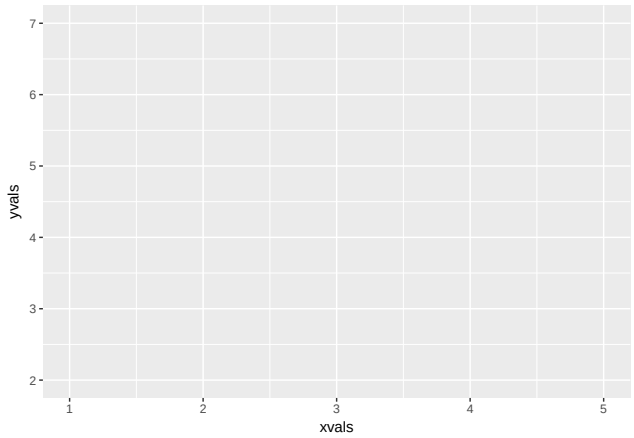


Figure 4.2: A ggplot2 plot without geoms.

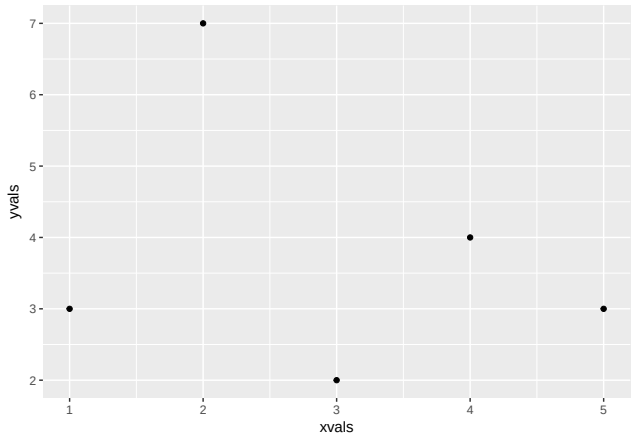


Figure 4.3: A ggplot2 plot with the point geom.

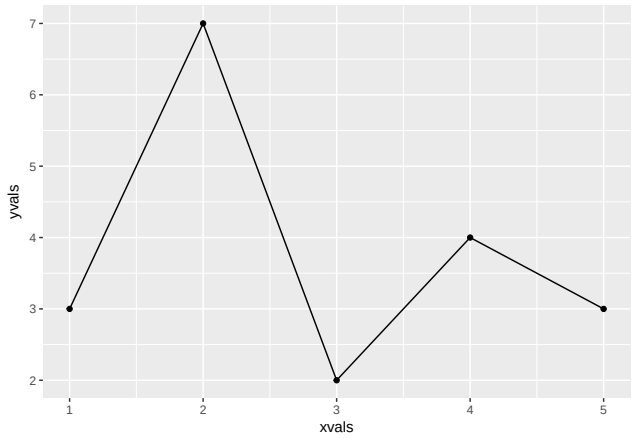


Figure 4.4: A ggplot2 plot with point and line geoms.

```
collatz <- function(n, limit = 10000) {
  # record initial numer because will change n
  initial <- n
  numbers <- numeric(limit)
  counter <- 0
  while ( n > 1 & counter < limit) {
    counter <- counter + 1
    numbers[counter] <- n
    n <- collatzRule(n)
  }
  howMany <- min(counter, limit)
  steps <- 1:howMany
  cat("The Collatz sequence has ", howMany, " elements.\n", sep = "")
  show <- readline("Do you want to see it (y/n)?  ")
  if ( show == "y" ) {
    print(numbers[steps])
  }
  # use initial value to make plot title:
  plotTitle <- paste0("Collatz Sequence for n = ", initial)
  # make the plot
  p <- ggplot(mapping = aes(x = steps, y = numbers[steps])) +
    geom_point() + geom_line() +
    labs( x = "Step", y = "Collatz Value at Step",
          title = plotTitle)
  # print it to the Plot Window
  print(p)
}
```

Try this version a few times, like so:

```
collatz(257)
```

It's quite remarkable how much the sequence can rise and fall before hitting 1.

One final consideration is validation of user-input. we don't want the user to get strange errors if she fails to enter a number worth looking at—a whole number that is greater than 1.

One possible approach is to attempt to coerce the user's input into an integer, using the `as.integer()` function:

```
as.integer(3.6) # will round to nearest integer
```

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

```
as.integer("4") # will convert string to number 4
```

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

```
as.integer("4.3") # will convert AND round
```

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

```
as.integer("hello") # cannot convert to integer
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

In the last example, the result is `NA`, and a cryptic warning was issued. In order to keep the warning from the user, we should wrap any call to `as.integer()` in the suppressWarnings()‘ function.

Let’s try out a piece of code that implements validation:

```
n <- -2
n <- suppressWarnings(as.integer(n))
isValid <- !is.na(n) && n > 1
  if (!isValid ) {
    stop("Need an integer bigger than 1.  Try again.")
  }
```

```
## Error: Need an integer bigger than 1.  Try again.
```

The `stop()` function throws a true error that prevents any further code from being executed, but allows us to write a custom, friendly error-message. The goal of validation is for the user to receive clear and helpful error messages, rather than obscure, technical messages from the depths of R.

Another issue worth considering is that our collatz() function depends on the previously-defined function `collatzRule()`. In order for it to work correctly, R would need to be able to find the name `collatzRule` somewhere on the search path. If the user hasn’t already defined the `collatzRule()` function, then a call to ‘collatz() will fail.

One solution is simply to remind users of the need to define both `collatzRule()` and `collatz()` prior to running `collatz()`. Another solution—perhaps the kinder one—is to define the `collatzRule()` function in the body of `collatz()`. Let’s adopt this approach.

The final version of the `collatz()` function appears below.

```
collatz <- function(n, limit = 10000) {
  # validation:
  n <- suppressWarnings(as.integer(n))
  isValid <- !is.na(n) && n > 1
  if (!isValid ) {
    stop("Need an integer bigger than 1.  Try again.")
  }
  # define collatzRule:
  collatzRule <- function(m) {
    if ( m %% 2 == 0) {
      return(m/2)
    } else {
      return(3*m + 1)
    }
  }

  # On with the show!
  # Record initial number because we will change it:
  initial <- n
  numbers <- numeric(limit)
```

```
  counter <- 0
  while ( n > 1 & counter < limit) {
    counter <- counter + 1
    numbers[counter] <- n
    n <- collatzRule(n)
  }
  howMany <- min(counter, limit)
  cat("The Collatz sequence has ", howMany, " elements.\n", sep = "")
  show <- readline("Do you want to see it (y/n)?  ")
  if ( show == "y" ) {
    print(numbers[1:howMany])
  }
  plotTitle <- paste0("Collatz Sequence for n = ", initial)
  steps <- 1:howMany
  p <- ggplot(mapping = aes(x = steps, y = numbers[1:howMany])) +
    geom_point() + geom_line() +
    labs( x = "Step", y = "Collatz Value at Step",
          title = plotTitle)
  print(p)
}
```

Glossary

Flow Control The collection of devices within a programming language that allow the computer to make decisions and to repeat tasks.

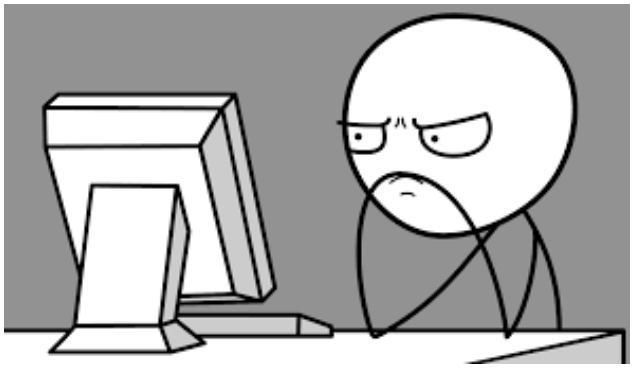
Condition A Boolean expression that commences an `if` or `while` statement. If the condition evaluates to `TRUE`, then the code in the body of the statement will be executed. Otherwise the code will be ignored.

Index Variable The variable in a `for` loop that takes on each of the values of the iterable in succession.

Iterable The vector that provides the sequence of values for the index variable in a `for` loop.

Validation The process of checking user input and rejecting it—usually with helpful suggestions—if it is not of the correct form.

Exercises



1. The *absolute value* of a number x is defined to be the number x itself if $x \geq 0$, whereas it is the opposite of x if $x < 0$. Thus:

   - the absolute value of 3 is 3;
   - the absolute value of -3 is -(-3), which is 3;
   - the absolute value of -5.7 is -(-5.7), which is 5.7
   - the absolute value of 0 is 0.

   The absolute value is important enough that R provides the `abs()` function to compute it. Thus:

```
abs(-3.7)
```

```
## [1] 3.7
```

   Write a function called `absolute()` that computes the absolute value of any given number `x`. Your code should make no reference to R's `abs()`.

   Small Bonus: Write the function so that it follows the vector-in, vector-out principle, that is: when it is given a vector of numbers it returns the vector of their absolute values, like this:

```
vec <- c(-3, 5, -2.7)
absolute(vec)
```

```
## [1] 3.0 5.0 2.7
```

2. Write a function called `pattern()` that, when given any character x and any positive number n, will print the following pattern to the console:

   - a line with just one x,
   - another line with two x's,
   - another line with three x's,
   - and so on until …
   - a line with n x's, and then
   - another line with $n - 1$ x's,
   - and so on until …
   - a line with just one x.

   The function should take two arguments:

   - `char`: the character to repeat. The default value should be `"*"`.
   - `n`: the number of characters in the longest, middle line. The default value should be 3.

   Typical examples of use should be as follows:

```
pattern()
```

```
## *
## **
## ***
## **
## *
```

```
pattern(char = "y", n = 5)
```

```
## y
## yy
## yyy
## yyyy
## yyyyy
## yyyy
## yyy
## yy
## y
```

3. Write a function called `beerTune()` that prints out the complete lyrics to the song *Ninety-Nine Bottles of Beer on the Wall*. You'll recall that the song goes like this:

   99 bottles of beer on the wall,

   99 bottles of beer!

   Take one down and pass it around:

   98 bottles of beer on the wall.

   98 bottles of beer on the wall,

   98 bottles of beer!

   Take one down and pass it around:

   97 bottles of beer on the wall.

   ...

   1 bottle of beer on the wall.

   1 bottle of beer!

   Take it down and pass it around:

   No more bottles of beer on the wall.

   Make sure to get the lyrics exactly right. For example, it's "1 bottle", not "1 bottles".

4. Recall the function `madhavaPI()` from Section 3.4.1. Use this function to write a function called `madhavaGraph()` that will do the following: given a number n, the function uses **ggplot2** to produce a line graph of the first n approximations to π using the initial terms of the Madhava series. The plot should be a line graph similar to the one produced by the `collatz()` functions from this Chapter. The function should take a single argument `n`, whose default value is 10. It should validate the input: if the number entered is not at least 1, then the function should explain to the user that the he/she must enter a positive number, and then stop.

   Here is an example of how the function should work:

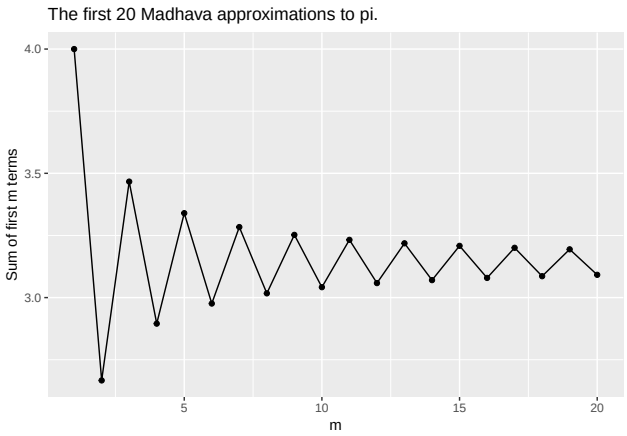


Figure 4.5: This is how the output of the madhavaGraph function should look.

```
madhavaGraph(n = -3)
```

```
## You need to enter a positive integer.  Try again!
```

Here is another example:

```
madhavaGraph(n = 20)
```

The output should be as in Figure 4.5.

5. (*) **The Subtraction Game.** In this game there are two players, A and B, and a pile of n pebbles. The players take turns removing pebbles from the pile. On each turn, a player can take either one or two pebbles. The players who takes the last pebble wins the game.

   It turns out that one of the players has a winning strategy. If the initial number of pebbles is a multiple of 3, then player who goes second has a winning strategy, whereas if the initial number of pebbles is not a multiple of 3 then the player who goes second has the winning strategy.

   The idea for the winning strategy comes from the following observations:

   - If there are 3 pebbles left and it's the other player's turn, then you will win. Why? Because after the other player removes pebbles there will be either 1 or 2 pebbles left. In either case you will be able to take the last pebble.
   - If there are 6 pebbles left and it's the other player's turn, then you will win. Why? Because after the other player removes pebbles there will be either 4 or 5 pebbles left. In either case on your turn you will be able to reduce the number of pebbles to 3. The game is now in the state of the previous bullet item, where we know that you will win.
   - If there are 9 pebbles left and it's the other player's turn, then you will win. Why? Because after the other player removes pebbles there will be either 7 or 8 pebbles left. In either case on your turn you will be able to reduce the number of pebbles to 6. The game is now in the state of the previous bullet item, where we know that you will win.
   - And so on, for 12 left, 15 left, 18 left, etc.

   As long as the number of pebbles left is a multiple of 3 and it's the other player's turn, you will win!

   In this problem your task is to write a function called `subtraction()` that will play the Subtraction Game with a user. The function should take two parameters:

   - `n`: the number of pebbles to begin with. The default value should be 12.

- `userFirst`: a logical parameter that indicates whether the user or the computer plays first. The default value should be `TRUE`, meaning that the user goes first.

Each time the computer plays it should announce how many pebbles it removed, and how many are left. When there are no pebbles left the computer should say who won and then quit the function.

The function should play optimally for the computer:

- if at the outset the computer has a winning strategy, then the computer should follow it and win.
- if at the outset the user has a winning strategy then the computer watch for opportunities to win if the user departs from her winning strategy.

Chapter 5

Turtle Graphics

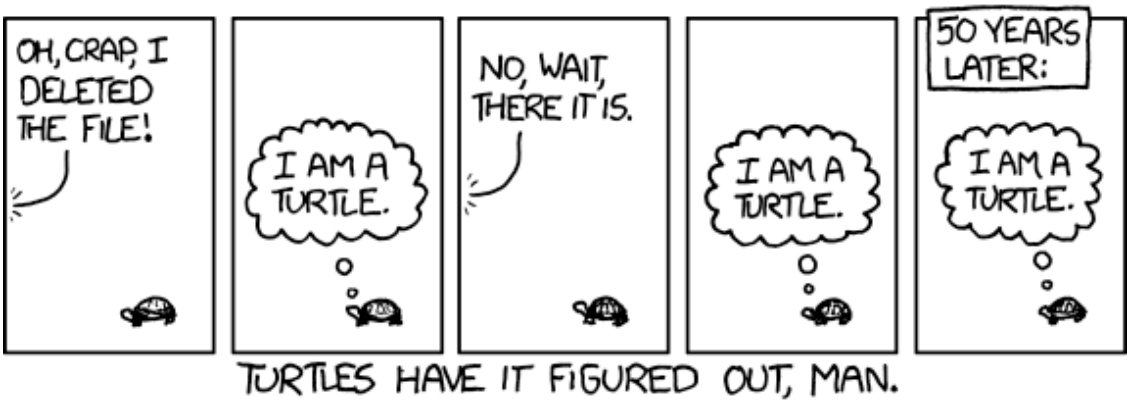


Figure 5.1: Turtles, by xkcd.

In this chapter we will practice our knowledge of R—and of basic programming concepts—in the context of a special R-package for graphics: package **TurtleGraphics** (Cena et al., 2014). Many of the examples from this Chapter are drawn from the Vignette for the package.[1]

[1]Turtle Graphics itself is not original to R. It was developed in the 1960's by Seymour Papert for use in the LOGO programming language. LOGO is a general-purpose programming language but has been primarily used to teach programming concepts to children. Nevertheless, grown-ups enjoy playing with Turtle Graphics so much that implementations of the system are now found in several major "professional-grade" programming languages. For more information about Turtle Graphics, consult Papert (1993).

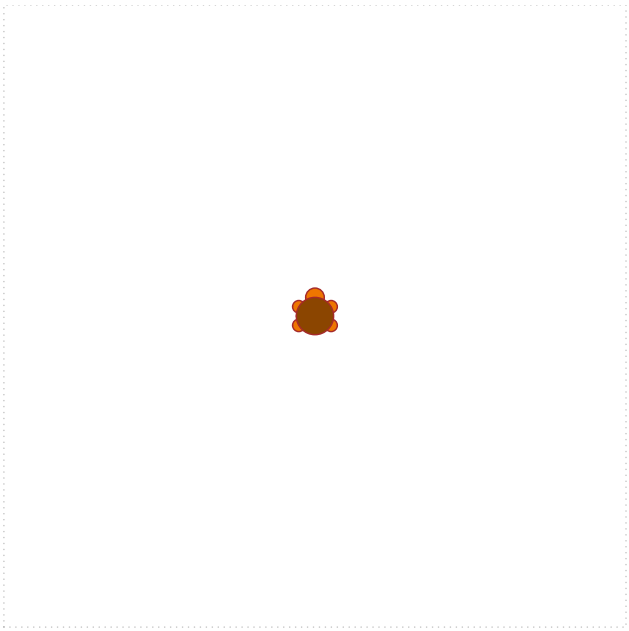


Figure 5.2: Initialized Turtle

5.1 Basic Movements

First, we begin by loading the package:

```
library(TurtleGraphics)
```

In order to create a Turtle Graphics scenario, call the function `turtle_init()`. You get the plot shown in Figure 5.2.

```
turtle_init()
```

By default the turtle is positioned in the middle of a square of dimensions 100 units by 100 units. (These dimensions can be changed, as we will see later on.)

You can get the turtle's position at any time:

```
turtle_getpos()
```

```
##  x  y
## 50 50
```

The turtle also begins facing North. This is considered to be angle 0, as you can tell by asking for the current angle of the turtle:

```
turtle_getangle()
```

```
## angle
##     0
```

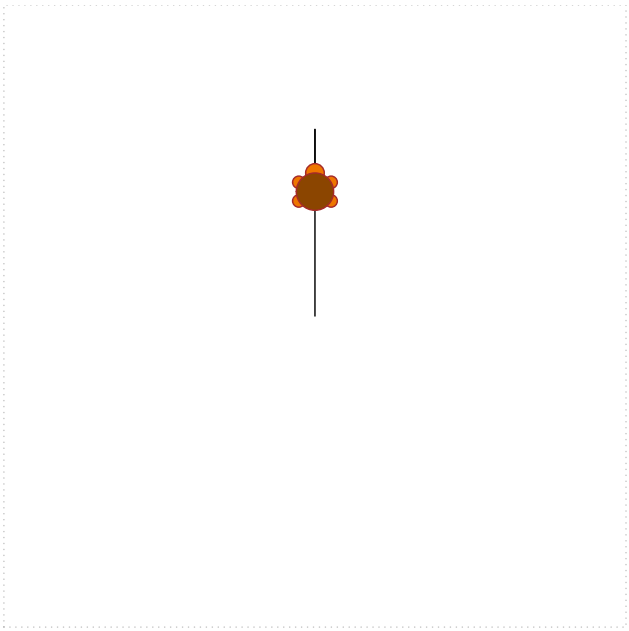


Figure 5.3: First Movements

Now let's make the turtle move. If you are following along on your own computer, it's best to run the lines of code one at a time, so you can see the effect of each command. (If you run multiples lines, you'll only see the graph produced by the final line.)

```
turtle_forward(dist = 30)
turtle_backward(dist = 10)
```

The result appears in Figure 5.3. Next we'll add a little triangle:

```
turtle_right(90)
turtle_forward(10)
turtle_left(angle = 135)
turtle_forward(14.14)
turtle_left(angle = 90)
turtle_forward(14.14)
turtle_left(angle = 135)
turtle_forward(10)
```

You can see the triangle in Figure 5.4.

The turtle is set in the "down" position, so that it leaves a trace out the path that it follows. You can avoid the trace by pulling the turtle "up" with `turtle_up()`. Whenever you want to restore the tracing, call `turtle_down()`. See Figure 5.5 for the results of the following code.

```
turtle_up()  # stop tracing
turtle_right(angle = 90)
turtle_forward(dist = 10)
turtle_right(angle = 90)
turtle_forward(dist = 17)
turtle_down()  # start tracing again
```

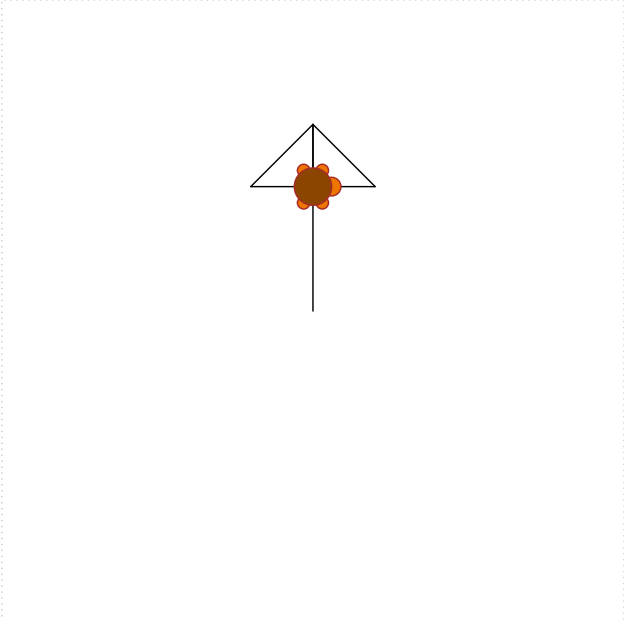


Figure 5.4: Adding a Triangle

```
turtle_left(angle = 180)
turtle_forward(dist = 34)
```

You can change the color of the lines your turtle draws:

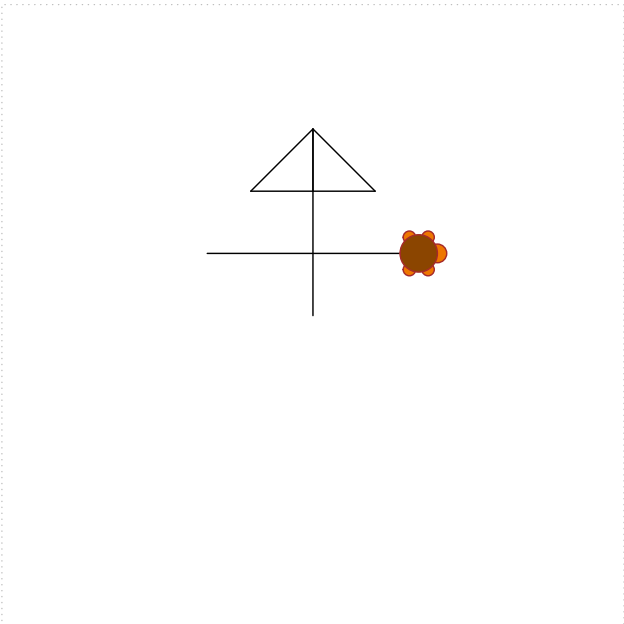


Figure 5.5: We moved the turtle around in the up position, then put the turtle down and traced out a line segment.

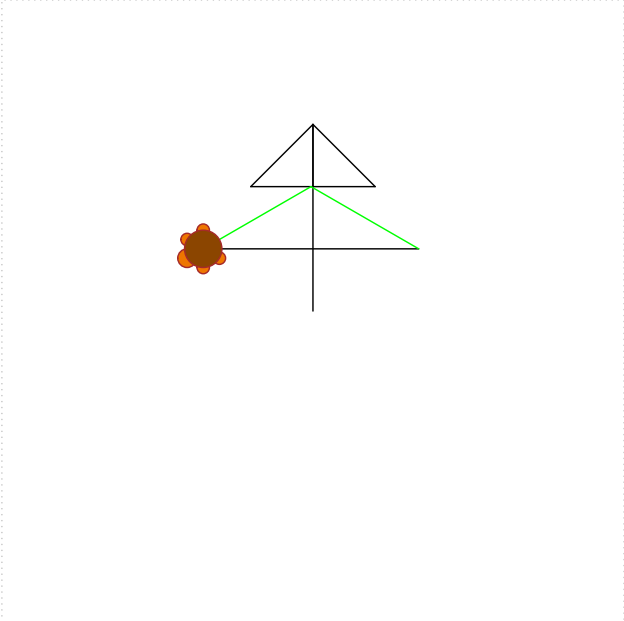


Figure 5.6: The graph after hiding, moving and showing.

```
turtle_col(col = "green")
```

In R here are many, many colors to choose from, and 657 of them even have names. To view them, use:= the `colors()` function:

```
colors()
```

You can also hide your turtle, and show it again any time you like. See Figure 5.6 for the results of the following code.

```
turtle_hide()
turtle_left(angle = 150)
turtle_forward(dist = 20)
turtle_left(angle = 60)
turtle_forward(dist = 20)
turtle_show()
```

Finally, you can choose the type of line your turtle draws, and the width of the line. See Figure 5.7 for the results of the following code.

```
turtle_left(angle = 150)
turtle_lty(lty = 4)
turtle_forward(dist = 17)
turtle_lwd(lwd = 3)
turtle_forward(dist = 15)
```

Note: you can learn more about `lty` and `lwd` with `help(par)`.

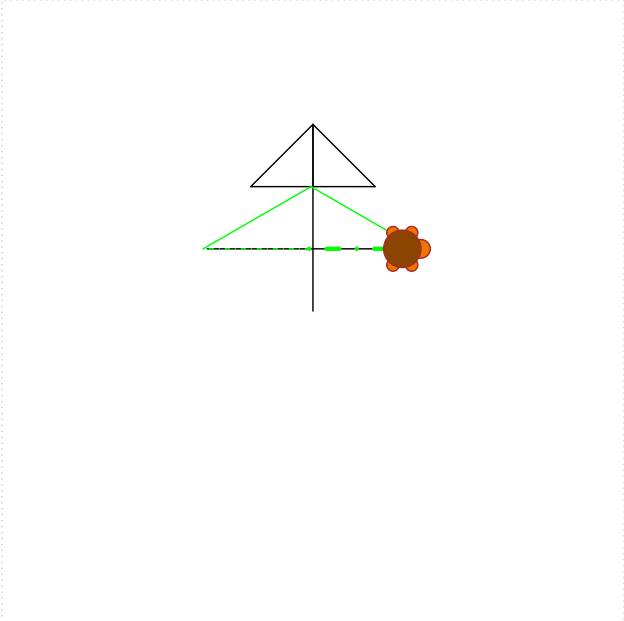


Figure 5.7: Choosing line-type and line-width.

5.2 Making Many Movements: An Introduction to Looping

Eventually we want to make some complex figures that require many movements on the part of the turtle. In order to make these go faster, we can turn off some of the turtle graphing by wrapping the desired movements in `turtle_do()`. See Figure 5.8 for the results of the following code.

```
turtle_init()
turtle_do({
  turtle_move(10)
  turtle_turn(45)
  turtle_move(15)
})
```

(`turtle_turn()` turns to the left by default.) Of course, for such a small number of movements using `turtle_do()` does not matter much, but we will practice using it for a bit.

How might we make a square? The following code offers one way to do it. See Figure 5.9 for the results.

```
turtle_init()
turtle_do({
  turtle_move(20)
  turtle_right(90)
  turtle_move(20)
  turtle_right(90)
  turtle_move(20)
  turtle_right(90)
  turtle_move(20)
  turtle_right(90)
})
```

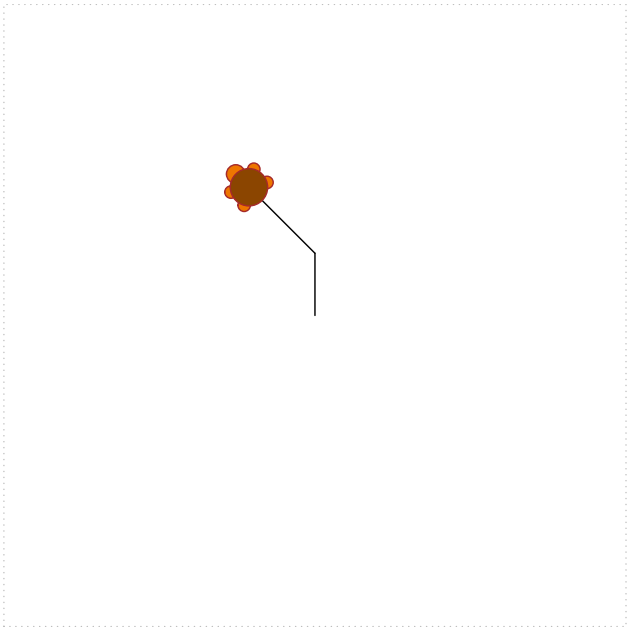


Figure 5.8: After final movement.

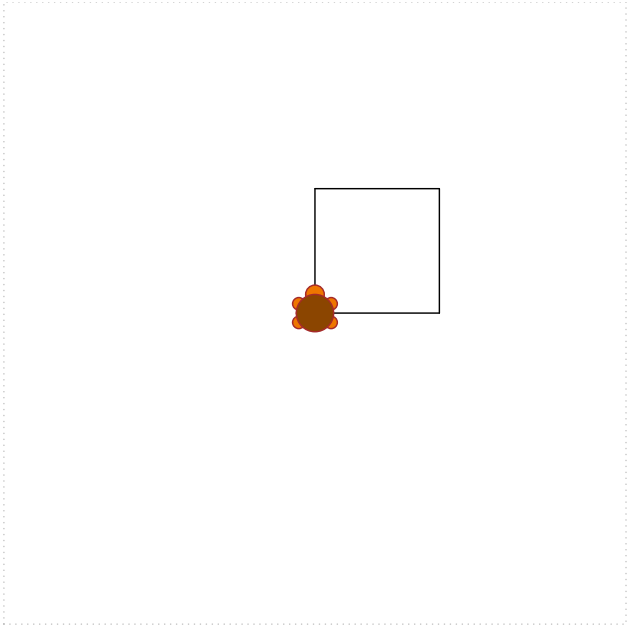


Figure 5.9: Making a square.

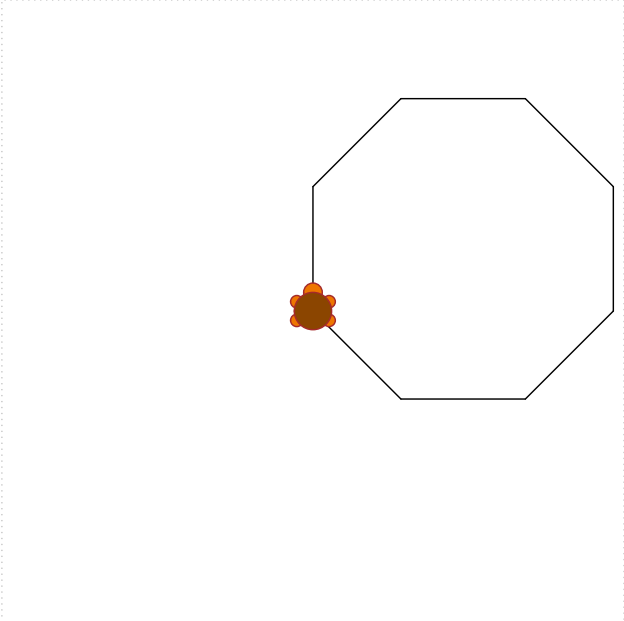


Figure 5.10: Making an octagon.

This is a bit repetitious. Surely we can take advantage of the fact that there is a clear pattern to the turtle's movements. A `for`-loop seems called for, as in the following code to build the square:

```
turtle_init()
turtle_do({
  for(i in 1:4) {
    turtle_forward(dist = 20)
    turtle_right(angle = 90)
  }
})
```

As we learned in Chapter 4, the more you need to repeat a particular pattern, the more it makes sense to write your code with a loop. Suppose, for example, that you decide to make regular octagons. A regular octagon has eight sides, and you turn 45 degrees after drawing each side. You can do this easily by modifying the square-code as follows (see Figure 5.10 for the results):

```
turtle_init()
turtle_do({
  for(i in 1:8) {
    turtle_forward(dist = 20)
    turtle_right(angle = 45)
  }
})
```

You can even make many small turns, so that the resulting figure starts to look like a circle (see Figure 5.11 for the results):

```
turtle_init()
turtle_setpos(x = 30, y = 50)
```

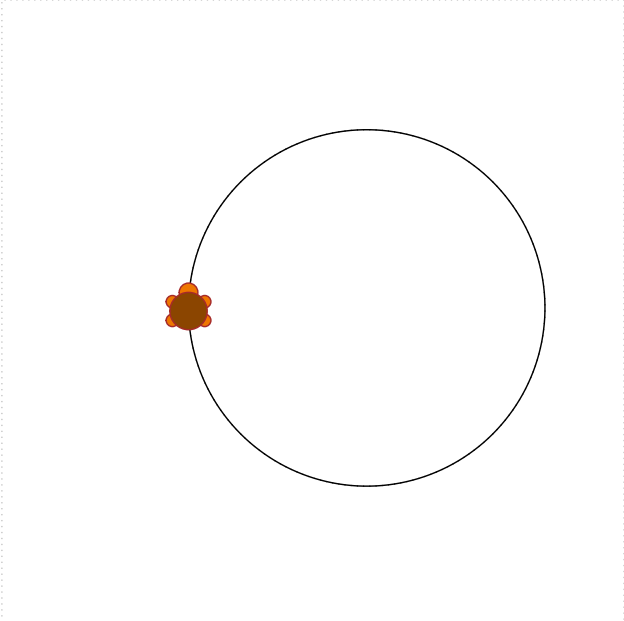


Figure 5.11: Making a circle.

```
turtle_do({
  for(i in 1:180) {
    turtle_forward(dist = 1)
    turtle_right(angle = 2)
  }
})
```

Notice that in the above code the turtle was initially set a bit to the left of center, so that the resulting circle would be situated close to middle of the region.

If you allow the index variable to be involved in the computations in the body of the loop then you can start making more complex figures. For example, here is the code for a spiral (see Figure 5.12 for the results):

```
turtle_init(width = 150, height = 150, mode = "clip")
turtle_do({
  turtle_right(90)
  for (i in 1:720) {
    turtle_left(1)
    turtle_forward(i/720)
  }
})
```

The turtle turns one degree every time R goes through the loop, but the amount it travels forward ($i/720$) increases as the index variable `i` increases.[2]

[2]Another thing to notice is that we set the width and height of the region ourselves, so that the spiral would fit into it. We also set `mode` to `clip` rather then leaving it at its default value of `error`. With `mode = "clip"`, R won't throw an error message at you when the turtle moves outside of its region. Clip-mode is very handy when you are developing a graph and don't know in advance precisely where the turtle will go.

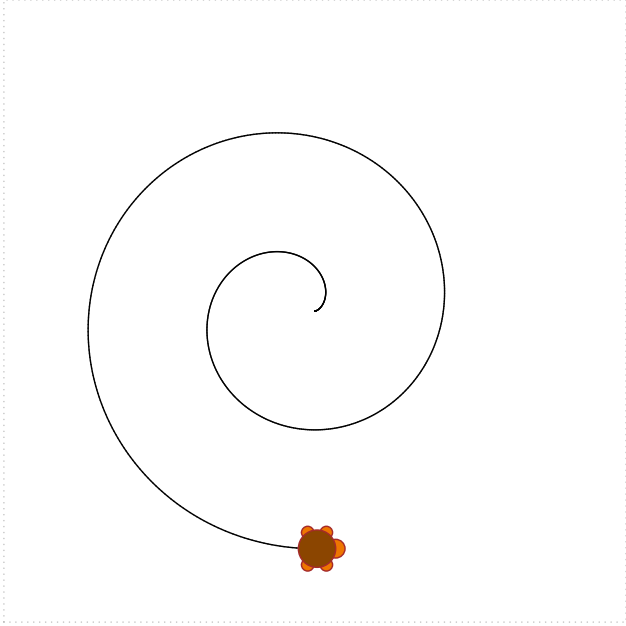


Figure 5.12: Using a loop to make a spiral.

5.3 Writing Turtle Functions

Once you have designed some shapes that you think you might want to draw again, you should write them up as functions. Here for example, is the code for a function that makes squares:

```
turtle_square <- function(side) {
  for (i in 1:4) {
    turtle_forward(side)
    turtle_right(90)
  }
}
```

Note that the user can vary the length of a side.

Functions are nice, but note that `turtle_do()` cannot be used freely inside of them. For example, suppose you were to write the square-making function as follows:

```
turtle_square <- function(side) {
  turtle_do({
    for (i in 1:4) {
      turtle_forward(side)
      turtle_right(90)
    }
  })
}
```

Attempting to use the function would result in an error:

```
turtle_init()
turtle_square(50)
```

```
Error in stopifnot(is.numeric(distance),
  length(distance) == 1, is.finite(distance)) :
 object 'side' not found
```

`turtle_do()` cannot find the side-length argument that was passed into the function. Although `turtle_do()` can appear inside of functions, it has to work on literal values, not on variables.

5.4 Random Moves

So far our turtle has moved in very regular and disciplined ways. It's time to break the pattern, a bit. R has a quite a few functions to generate numbers that look "random"; we will use some of these functions to make the turtle move about randomly.

5.4.1 Sampling from a Vector

You have already met the `sample()` function (see Section 4.2.1). Let's take a closer look at it.

`sample()` makes a random choice from a given vector. From the R-help we read that the general form of a call to sample is as follows:

```
sample(x, size, replace = FALSE, prob = NULL)
```

In the above call:

- `x` is the vector from which we wish to sample (R refers to it as the "population");
- `size` is the number of random samples we want;
- `replace` says whether or not to replace each member of the population after we have sampled it.
- `prob` specifies the desired probability for each member of the population to be chosen.

A few examples will help us understand how the arguments work:

```
vec <- 1:10 # we'll sample from this vector
sample(vec, 1)
```

```
## [1] 7
```

We got only one number because we set `size` to 1. Every element in `vec` had an equal chance of being the element selected. This time we got 7, but if you were to run the function again for yourself your results would probably be different.

Let's sample 10 numbers from `vec`:

```
sample(vec, 10)
```

```
##  [1]  3  5  2  4  7  6  9  1  8 10
```

Because `replace` was left at its default value of `FALSE`, R did not replace numbers after pulling them from the `vec`. After each sample, the remaining numbers all had the same chance to be picked next. Setting `size` to the length of `x` and keeping `replace = FALSE` therefore has the effect of randomly shuffling the elements of `x`.

Of course when `replace = FALSE` any attempt to sample more than the number of elements of `vec` will result in an error:

```
sample(vec, 11)
```

```
Error in sample.int(length(x), size, replace, prob) :
cannot take a sample larger than the
population when 'replace = FALSE'
```

When we set `replace = TRUE` then each selected element is returned to the population. At any stage, the chance for a given member of the population to be the one selected next is the same—no matter how many times that member has already been selected. Thus, when `replace = TRUE` you are liable to see repeats:

```
sample(vec, 20, replace = TRUE)
```

```
##  [1]  3  8  8  8  8  9  5  6  7  1  4  8  4  1 10  6  3  3  1  9
```

When the `prob` parameter is left at its `NULL` value, R gives each member of the population the *same* chance to be the member that is selected. It is possible to adjust the probabilities of selection by setting `prob` to a vector of probabilities (one for each corresponding member of `x`). Thus, suppose we want to select 20 numbers from `vec`, according to the following the probabilities:

- 5% chance of selection, for each number from 1 to 8;
- 30% chance for 9 to be selected;
- 30% chance for 10 to be selected.

Then we can call `sample()` like this:

```
sample(vec, 20, replace = TRUE,
       prob  = c(rep(0.05, 8), 0.30, 0.30))
```

```
##  [1]  9  1  9 10  9  6  8  9  4  7 10 10  9  2  6  9  5  9 10 10
```

Notice that we a majority of 9's and 10's: this was fairly likely to occur since each selection had a 60% chance of turning out to be 9 or 10.

5.4.2 Application: a Bouncing Turtle

Let's apply `sample()` to design a scenario in which the turtle moves a fixed amount at each step, but the direction—north, east, south, or west—is completely random. When the turtle reaches the boundary of its domain, however, we would like it to "bounce back": i.e., take a step in the direction opposite to the step that brought it to the boundary. We will also query the user prior to each step, asking if he/she wants to see another move. This not only allows the user to decide when to end the scenario; it also permits the user to see where the turtle is after each step.

One possible implementation is as follows:

```
turtle_bounce <- function(side = 60, step= 10) {
  if ( (side/2) %% step != 0 ) {
    stop("Side-length divided by two must be a multiple of step.")
  }
  bounds <- c(0, side)
  turtle_init(side, side, mode = "clip")
```

```
  origin <- turtle_getpos()
  cp <- turtle_getpos()
  repeat {
    move <- readline(prompt = "Go Again? (enter q to quit):  ")
    if ( move == "q") break
    x <- cp["x"]
    y <- cp["y"]
    if (x %in% bounds | y %in% bounds) {
      angle <- 180
    } else {
      angle <- sample(c(0,90,180,270), 1)
    }
    turtle_right(angle)
    turtle_forward(step)
    cp <- round(turtle_getpos(), 0)
    print(cp)
  }
  cat("All done!")
}
```

Play the game a few times, to get a feel for how it works:

```
turtle_bounce(60, 15)
```

Let's examine the code a bit more closely.

The definition indicates that there are two parameters: `side` and `step`.

- The `side` parameter gives the dimensions of the Turtle's field. Thus if `side` were set to 60—which is the default—then the field would be a 60-by-60 square, with the origin $(0, 0)$ in at lower-left corner and the point $(60, 60)$ at the upper-right corner. When the turtle is initialized it will appear in the middle of the square, at the point $(30, 30)$.
- `step` specifies how many units the turtle will move at each step. In this analysis we will assume that `step` is set to 15.

Inside the function, we begin with a bit of input-validation:

```
if ( (side/2) %% step != 0 ) {
    stop("Side-length divided by two must be a multiple of step.")
  }
```

Remember that the turtle will start at $(\texttt{side}/2, \texttt{side}/2)$ and will move `step` each time. If `side`/2 is not evenly divisible by `step` then the turtle would be able to go from inside its field to outside in a single step. We don't want that to happen so we stop the user if the remainder after dividing `side`/2 by `side` is anything other than zero.

If the input is OK, then we set up a vector `bounds` that records the smallest and largest possible values for the x and y coordinates of the turtle:

```
bounds <- c(0, side)
```

Next, we initialize the turtle in the middle of the field and record its initial position in the vector `cp`:

```
turtle_init(side, side, mode = "clip")
origin <- turtle_getpos()
cp <- turtle_getpos()
```

(You can think of `cp` as short for: "current position".)

Next, we enter a `repeat`-loop. Inside the loop we begin with:

```
move <- readline(prompt = "Go Again? (enter q to quit):  ")
if ( move == "q") break
x <- cp["x"]
y <- cp["y"]
```

We first asked the user if she wanted to quit. If she enters "q" then we'll break out of the loop and end the scenario. If she enters anything else (including just pressing Enter) then we record the x and y coordinates of the turtle's current position in the vectors `x` and `y` respectively.

Our next task is to determine how the turtle should move:

```
if (x %in% bounds | y %in% bounds) {
      angle <- 180
    } else {
      angle <- sample(c(0,90,180,270), 1)
    }
```

If the turtle is at a boundary (either `x` equal to 0 or 60 or `y` equal to 0 or 60) then we need to "bounce back". This corresponds to making the turtle turn right by 180 degrees and then step. On the other hand if the turtle is not at a boundary then the direction of the turtle should be random, so we should have it turn right by either 0, 90, 180 or 270 degrees, with each possibility being equally likely. This is accomplished with the above call to the `sample()` function.

Having determined the amount by which to turn prior to the next step, we then have the turtle turn that amount and take the step:

```
turtle_right(angle)
turtle_forward(step)
```

Finally, we set `cp` to the new position of the turtle, and print that position out to the console for the user to see:

```
cp <- round(turtle_getpos(), 0)
print(cp)
```

Note that we rounded off the position to the nearest while number. This was done because the authors of the **TurtleGraphics** package use floating point arithmetic for their numerical operations, so sometimes the computed positions differ from whole numbers by a very tiny amount.

We then repeat the loop.

5.4.3 Uniform Random Numbers

`sample()` picks an element from a finite population. Sometimes, though, we want R to give the impression that it has picked a *real number* at random out of a range of real numbers. This can be accomplished with the `runif()` function.

A call to `runif()` looks like this:

```
runif(n, min = 0, max = 1)
```

The idea is that R will produce `n` real numbers that have the appearance of having been drawn randomly from the interval of real numbers whose lower and upper bounds are specified respectively by `min` and `max`.

Thus, to get 10 "random" numbers that all lie between 0 and 1, you can leave `min` and `max` at their defaults and ask for:

```
runif(10)
```

```
##  [1] 0.646902839 0.394225758 0.618501814 0.476891136 0.136097186
##  [6] 0.067384386 0.129152617 0.393117930 0.002582699 0.620205954
```

5.4.4 Pseudo-Randomness and Setting a Seed

It's important to point out that R doesn't generate truly random numbers.[3] After all, R simply runs a computer which operates according to a set of completely-specified steps. Thus the random data generated by R and by other computer languages is often called *pseudorandom*. Although the functions for random-number generation have been carefully designed so as to follow many of the statistical laws we associate with randomness in nature, all of the pseudo-random output is determined by an initial value and a deterministic number-generating algorithm.

We actually have the ability to set the pseudorandom data ourselves. This is called *setting the random seed*. From any specified seed, the result of calls to R's random-data functions will be completely determined (although—just as in the case of "real" randomness—the output will still probably "look" random).

The `set.seed()` function will fix the random output. Try running the following two lines of code more than once:

```
set.seed(2025)
runif(10)
```

```
##  [1] 0.7326202 0.4757614 0.5142159 0.4984323 0.7802845 0.5042522 0.8984003
##  [8] 0.1278527 0.6446721 0.5695311
```

You will get the same output every time. If you change the argument of `set.seed()` to some other integer the output will probably change—but it will stay the same when you run the code *again* from that that new seed.

5.4.5 Application: a Drunken Turtle

We will now modify the previous scenario so that the turtle's motion will be almost completely random. Even though it will take the same-size step every time, the angle at which it steps will be completely random: any real number of degrees from 0 to 360. We will also show the user the position of the turtle at each step, and

[3] Indeed, philosophers of mathematics debate what randomness "really" is.

use the "distance formula" from high-school geometry to compute and display the current distance of the turtle from the place where it started.

```
turtle_drunk <- function(side, step) {
  turtle_init(side, side, mode = "clip")
  # save (side/2, side/2), the turtle's initial position:
  initial <- turtle_getpos()
  repeat {
    move <- readline(prompt = "Go Again? (enter q to quit):  ")
    if ( move == "q") break
    # pick a random angle to turn by:
    angle <- runif(1, min = 0, max = 360)
    turtle_left(angle)
    turtle_forward(step)
    # get new position, make it the current position:
    cp <- turtle_getpos()
    # print to console:
    print(cp)
    # determine distnce from initial position (round to 3 decimals):
    distance <- round(sqrt((cp[1] - initial[1])^2 + (cp[2] - initial[2])^2),3)
    # prepare message to console,and print it:
    message <- paste0("Distance from starting point is: ", distance)
    cat(message)
  }
  cat("All done!")
}
```

Try the game once or twice:

```
turtle_drunk(100, 5)
```

It is natural to wonder how likely the turtle is to wander back close to where it started, and to wonder how often that will happen. We will address questions like these in Chapter 6.

5.5 More Complex Turtle Graphs

Simple instructions, when combined with looping, can produce quite complex patterns. Consider the following process (with results shown in Figure 5.13):

```
turtle_init(1000, 1000, mode = "clip")
turtle_do({
  turtle_setpos(600,400)
  turtle_right(90)
  for (i in 1:2000) {
    turtle_right(i)
    turtle_forward(sqrt(i))
  }
})
```

You might enjoy figuring out why this pattern occurs. As you ponder this, it might help to construct a set of "ragged" spirals with somewhat larger steps, and pause at each step. Code like the following might be

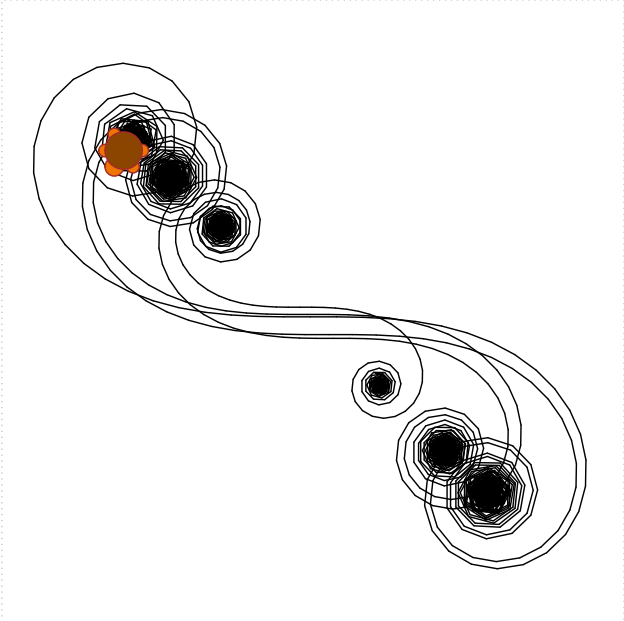


Figure 5.13: Galactic Zany!

useful:[4]

```
turtle_init(1000, 1000, mode = "clip")
turtle_do({
  i <- 1
  turtle_right(90)
  repeat {
    bidding <- readline("Proceed? (Enter q to quit) ")
    if ( bidding == "q") break
    turtle_right(i)
    turtle_forward(2*sqrt(i))
    cat(paste0("Turned ", i, " degrees,\n"))
    cat(paste0("stepped forward ", round(2*sqrt(i), 3), " units.\n"))
    cat("Turtle's current angle is: ", turtle_getangle(), " degrees.\n")
    i <- i + 20
  }
  cat("All done!")
})
```

[4]Also don't forget that every 360 degrees is a full turn around the circle, so when the turtle's angle is, say 720 degrees, it's the same as an angle of 360 degrees which is the same as an angle of 0 degrees. All three angles amount to the same direction.

Glossary

Pseudo-random Numbers A sequence of numbers generated by a computer procedure designed to make the sequence appear to follow statistical laws associated with random processes in nature.

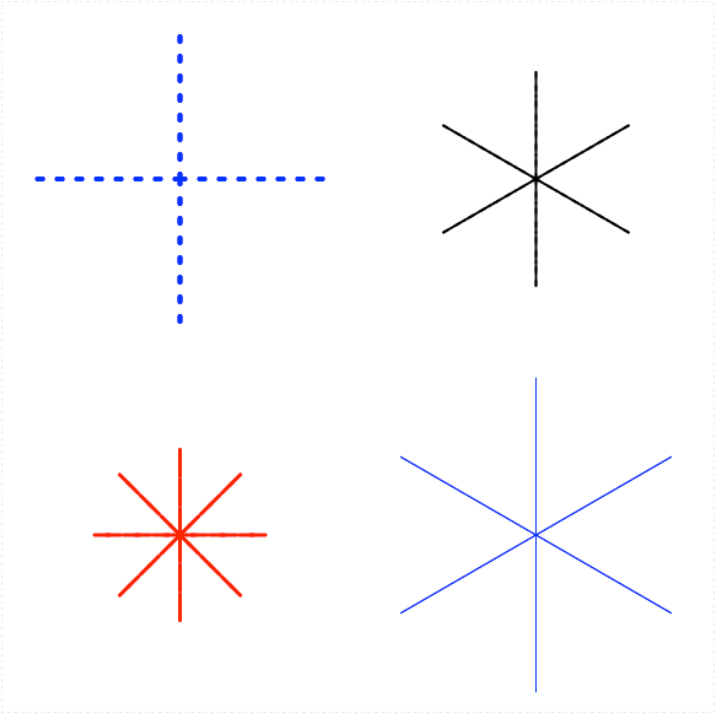
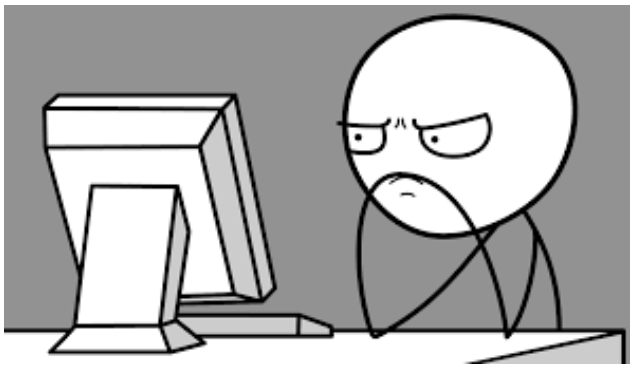


Figure 5.14: Sample Stars

Exercises



1. Write a function called `turtle_gon()` that draws a regular polygon. The user should be able to specify the side-length and the number of sides. Start with `turtle_init()` and use your function to draw a regular dodecagon (twelve sides) with each side having a side length of 15 units. Your figure should not stray outside of the turtle's field, so you may have to adjust the position of your turtle a bit prior to calling your function.

2. Write a function called `turtle_star()` that can make stars like the ones in Figure 5.14. The user should be able to specify:

   - the number of rays in a star (default is 6);
   - the length of the rays (default is 20);
   - the color of the rays (default is red);
   - the line-type of the rays (default is 1);
   - the line-width of the rays (default is 1).

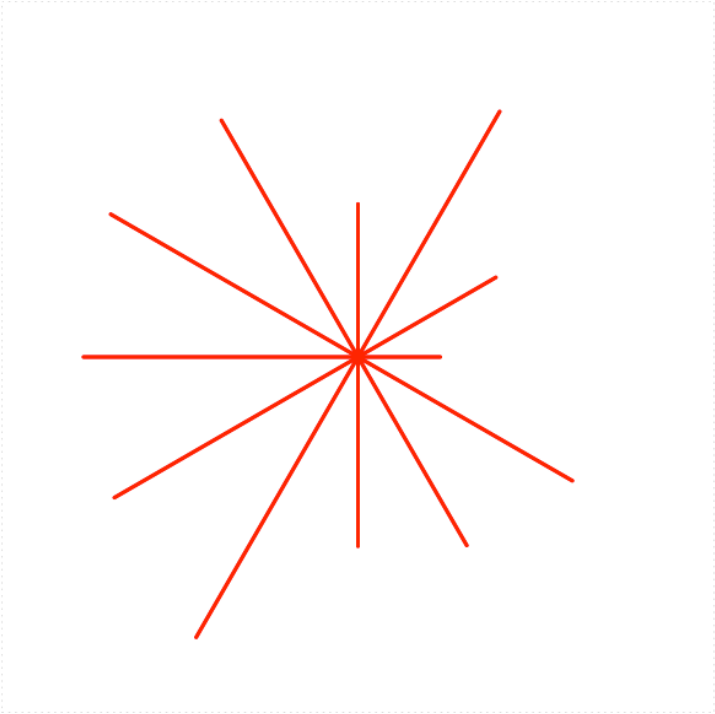


Figure 5.15: A star with rays of random lengths.

Starting with `turtle_init()`, use your function to create a star with 10 rays, each of length 20 units. The lines should be red and dashed. I'll leave the thickness up to you.

3. Make a new star function `turtle_rstar()` in which the lengths of the rays are not determined by the user but instead vary randomly from 5 to 25 units, as in Figure 5.15:

   The defaults for the other parameters should be the same as in the previous exercise. Starting from `turtle_init(50,50)` make a random star with 20 rays and a line-thickness of 3. Other parameters should be left at their default-values.

   Hint: You'll probably use a loop to draw each ray of the star. As you go through the loop, get a random ray-length using the `runif()` function:

```
randomLength <- runif(1, min = 5, max = 25)
```

   Then have the turtle make the ray by moving forward and backward by that amount.

4. Make a new star function `turtle_rstarColors()` that behaves like `turtle_rstar()` except that instead of being determined by the user the ray-color varies randomly from one ray to another, as in Figure 5.16:

   Each ray should have a color drawn randomly from the vector of all colors given by `colors()`. Starting from `turtle_init(50,50)` make a random star with 20 rays and a line-thickness of 6. Other parameters should be left at their default-values.

   Hint: You'll probably use a loop to draw each ray of the star. As you go through the loop, get a new random color like this:

```
randomColor <- sample(colors(), size = 1)
```

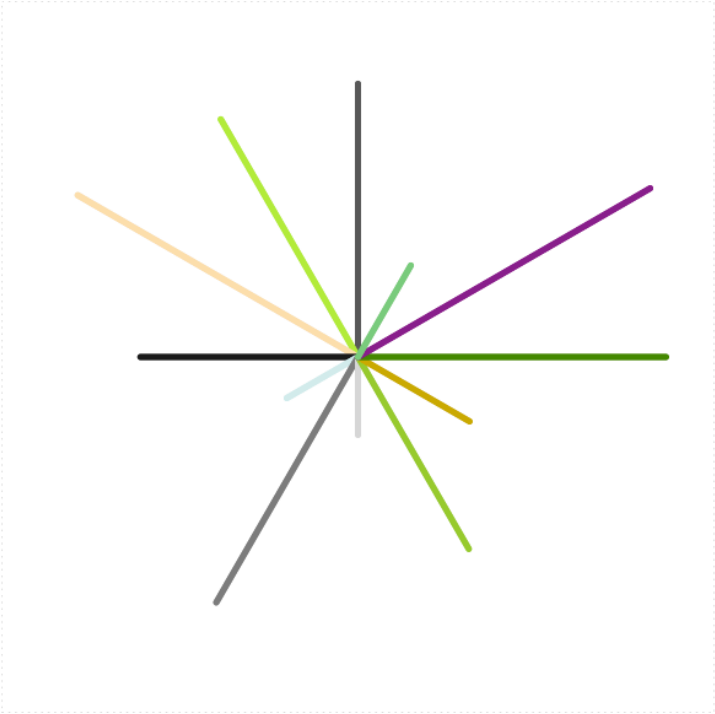


Figure 5.16: A star with rays of random lengths and random colors.

Then set the turtle's color accordingly:

```
turtle_col(col = randomColor)
```

Chapter 6

Simulation

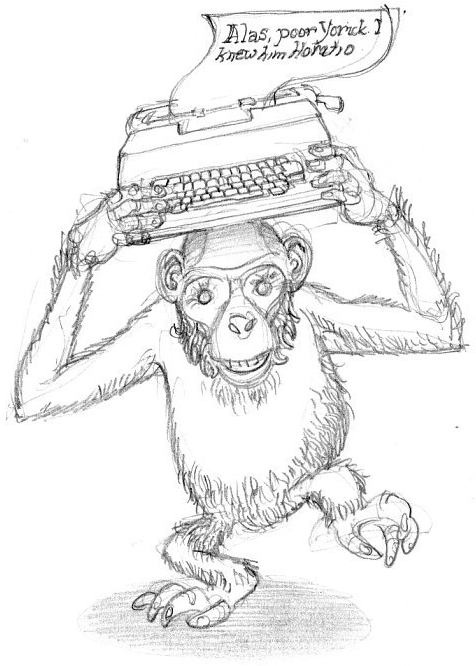


Figure 6.1: A monkey typing randomly manages to type out a line of Shakespeare. What is the chance that the monkey would produce this particular line? Source: http://www.gloryofkings.com/?p=189.

6.1 Probability and Random Variables

Your favorite college basketball team is in the NCAA Tournament. What is the chance that it will make the Final Four?

An insurance company provides flood insurance to homeowners. How much can the company expect to pay out in insurance claims in the coming year?

What is the chance of a large asteroid striking our planet in the next ten years? in the next ten million years?

All of the above questions involve random processes—some of them rather complex. It is the aim of this Chapter to apply our knowledge of vectors, functions and flow control to give approximate answers to questions involving chance processes like these.

Sometimes we will be interested in finding the *probability* that an event of interest occurs: for example, the event that your team makes the final Four, or the event that the Earth is struck by a large asteroid sometime in the next ten years.

One popular definition of probability goes as follows:

> If E is an event, then the probability that E occurs is the long-term proportion of times that the event occurs, if we could repeat the random process many, many times.

To be more precise, imagine that you can repeat the random process n times. (Imagine that your team gets into the NCAA Tournament n times, for example.) Each time, the event of interest will either occur or not occur. Count the number of times that the event occurs. Then divide by n. You now have the *proportion* of times that it occurred. Now imagine that n gets larger and larger. Our intuition says that the proportion of times that the event occurs will stabilize at some number between 0 and 1. This number is the *probability* that the event occurs. occurs.

We will also concern ourselves with *random variables*. A random variable is simply a number whose value is determined by the outcome of a chance process. The amount that the insurance company will pay out in the next year is an example of a random variable: its value depends on a complex chance process—how many homeowners experience a flood, how damaging each flood is, etc.

When it comes to random variables, we are often interested in what it might turn out to be *on average*. That is, suppose we could repeat the random process many, many times—say n times, where n is some large number. (Suppose that we could observe the insurance company for many, many years, with each year being the same in terms of how many people are insured, how much their houses are worth, what the climate is like, and so on.) Each time the process is repeated, we get a value for the random variable. We end up with n values. Now add up these values and divide by n. We have computed their average—the mean, as it is properly called. Now imagine that n could be larger and larger, without bound.

> The *expected value* of the random variable is the number that this average converges to.

In other words, the expected value of a random variable is what we expect to get on average, in the long run, if we could repeat the random process that produces the value many, many times.

6.2 Monte Carlo Simulation

Our definitions of probability and expected value both involved a limiting notion, namely: what would happen if you could somehow repeat the random process more and more times, without a bound on the number of repetitions. Accordingly, even if we find that we are unable to compute a probability or an expected value exactly with mathematics, we can still attempt to estimate it by **making the computer** repeat the random experiment many times, keeping track of the result of the experiment each time. This

technique is known as *Monte Carlo* simulation, after the famous Monte Carlo casino[1] in the Principality of Monaco.

In this section we will employ Monte Carlo simulation to estimate probability and expected value in a couple of simple examples.

6.2.1 Estimating a Probability

Consider a box that holds ten tickets. Three of them are labeled with the letter "a"; the rest are labeled "b":

```
tickets <- c(rep("a", 3), rep("b", 7))
tickets
```

```
##  [1] "a" "a" "a" "b" "b" "b" "b" "b" "b" "b"
```

We plan to draw one of these tickets at random from the box, with each ticket having the same chance to be the ticket selected. Three of the ten tickets are a's, so our intuition says that the probability of selecting an "a" ticket is 3 out of 10, or 0.3. Let's use Monte Carlo simulation to estimate the probability, and see if we get something close to 0.3.

In order to set up the simulation, we need a device for repeating the random process as many times as we would like. At each repetition, the outcome of the chance process should not depend on previous outcomes. We can accomplish this by using R s `sample()` function on the `tickets` variable. For example, if we would like to repeat the process of selecting a ticket twenty times, we could write the following code:

```
sims <- sample(tickets, size = 20, replace = TRUE)
sims
```

```
##  [1] "b" "b" "b" "a" "b" "b" "b" "b" "b" "b" "a" "a" "b" "b" "b" "b" "b"
## [18] "b" "a" "a"
```

Notice that we set `replace` to `TRUE`. This was important, since it guarantees that at each selection there are three "a"'s and seven "b"'s in the box, keeping the chance of getting an "a"-ticket the same each time.

We can summarize the results in a table using R's `table()` function:

```
table(sims)
```

```
## sims
##  a  b
##  5 15
```

Of course we don't care so much about the *number* of times we got an "a"-ticket. Instead we care about the *proportion* of times we did so. In order to convert the numbers in a table to proportions we can make the table an argument for the function `prop.table()`:

```
prop.table(table(sims))
```

```
## sims
##    a    b
## 0.25 0.75
```

[1] https://en.wikipedia.org/wiki/Monte_Carlo_Casino

Based on twenty repetitions of the random process, our estimate of the chance an "a"-ticket is 0.25. That's not so close to the true chance of 1/3. In order to obtain a more accurate estimate, we should increase the number of repetitions of the random process. Let's try again, with ten thousand repetitions:

```
sims <- sample(tickets, size = 10000, replace = TRUE)
prop.table(table(sims))
```

```
## sims
##     a     b
## 0.308 0.692
```

Our estimate is now a lot closer to the actual value of the probability.

6.2.2 Estimating an Expected Value

Imagine that you are about to play the following game: you will flip a fair coin twice.

- If you get Tails both times, you lose a dollar.
- If you get exactly one Head, nothing happens.
- If you get Heads both times, you win two dollars.

Let W be the number of dollars you will win.

W is a clearly a random variable: it's a number whose value—either -1, 0 or 2—depends on the outcome of the random process of flipping the fair coin twice. What is the expected value of W?

If we think about the probabilities involved then we can come up with a candidate for the expected value. When you flip a fair coin twice, there are four equally likely outcomes:

- Tails and then Tails ($W = -1$)
- Tails and then Heads ($W = 0$)
- Heads and then Tails ($W = 0$)
- Heads and then Heads ($W = 2$)

Hence you have:

- The chance that $W = -1$ is 0.25.
- The chance that $W = 0$ is 0.50.
- The chance that $W = 2$ is 0.25.

Hence the expected value should be the *weighted average*:

$$0.25 \times -1 + 0.50 \times 0 + 0.25 \times 2.$$

This works out to 0.25, or 25 cents.

We would like to see if Monte Carlo simulation can render an estimate that is close to this value.

For a simple game like ours where there are only a few possible outcomes for the random variable, it is still a good idea to use `sample()` to simulate the random process of playing the game. We only need to provide a vector of possible values for W to sample from, and a vector of probabilities for obtaining each of those possible values. The following code illustrates how we might simulate playing the game ten times:

```
w <- c(-1, 0, 2)
pw <- c(0.25, 0.50, 0.25)
winnings <- sample(w, size = 10, replace = TRUE, prob = pw)
winnings
```

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

The Monte Carlo estimate of the expected value of W is just the average of the winnings in our simulations:

```
mean(winnings)
```

```
## [1] 0.1
```

Of course, with such a small number of repetitions of the game we cannot expect the Monte Carlo estimate to be very accurate. Let's try again, but with ten thousand repetitions:

```
winnings <- sample(w, size = 10000, replace = TRUE, prob = pw)
mean(winnings)
```

```
## [1] 0.2435
```

Yes, that's much closer to our mathematically-computed value!

By the way, R can easily compute the weighted average itself:

```
sum(w*pw)
```

```
## [1] 0.25
```

6.2.3 The Law of Large Numbers

In our Monte Carlo simulations so far, we have seen that the more times we repeat the underlying random process, the closer our estimate is likely to be to the actual value, no matter whether we were estimating the probability of an event or an expected value for a random variable.

This is no accident: in fact, it is a consequence of a theorem in the subject of probability that is known as the *Law of Large Numbers*. We do not possess the mathematical machinery necessary to state the Law precisely—much less prove it—but we can take the rough statement given here as an assurance that the more repetitions we include in our simulation, the more accurate the resulting estimate is liable to be.

6.3 Example: Chance of a Triangle

In the examples considered so far, intuition or a simple calculation suggests what the exact value of the probability or expected value should be. The true power of the Monte Carlo simulation method shines forth in situations where it is difficult or impossible to compute a value mathematically.

Let's revisit an example from Section 4.2.3, where we considered whether or not three segments could be put together to form a triangle. We developed the following function to determine, from the lengths of each segment, whether or not a triangle is possible:

```
isTriangle <- function(x, y, z) {
  (x + y > z) & (x +z > y) & (y + z > x)
}
```

Recall also that this function applied to vectors of any length, so we can use it to decide about many triples of segments at once.

Let's now inject some chance variation into the situation. Suppose that you have a stick that is one unit in length: one foot, one meter, one yard—whatever. You plan to break the stick at two randomly selected points. Breaking the stick will yield three pieces. You wonder: what is the chance that these three pieces can form a triangle?

In an advanced probability course you can show that the probability of a triangle is exactly 1/4. Here we would like to estimate the probability with Monte Carlo simulation.

In order to carry out the simulation, we will need to develop code that produces a pair of break-points, any number of times that we like—let's think about ten times, to start.

This is not so difficult if we use the `runif()` function:

```
x <- runif(10)  # the first breaks
y <- runif(10)  # the second breaks
```

Next we must compute—for each pair of breaks—the lengths of the three segments that are produced. If we knew that the `x` break was always less than the `y` break, this would be easy:

- the leftmost piece would be `x` units long;
- the middle piece would be `y` minus `x` units long;
- the rightmost piece would be 1 minus `y` units long.

The problem is that a given element of the vector `y` can easily be less than than corresponding element of the vector `x`. When that happens, the pieces won't be as described in the bullet-ed items above.

We can solve this problem with the `pmin()` and `pmax()` functions that were introduced in Section 2.7.1.5:

```
a <- pmin(x, y)
b <- pmax(x, y)
side1 <- a     # leftmost
side2 <- b - a  # middle
side3 <- 1 - b # rigttmost
```

It is useful to write a helper-function that starts from the breaks, computes the sides, and then decides whether they form a triangle:

```
makesTriangle <- function(x, y) {
  a <- pmin(x, y)
  b <- pmax(x, y)
  side1 <- a
  side2 <- b - a
  side3 <- 1 - b
  isTriangle(x = side1, y = side2, z = side3)
}
```

Let's try it our for our repetitions

Table 6.1 shows the results. Based on our ten repetitions, we would estimate the probability of a triangle as 4/10.

Interestingly, R can compute the proportion of times that a triangle was formed, directly from the logical vector `triangle`. Look at the following code:

```
sum(triangle)
```

Table 6.1: Results of ten repetitions of breaking a unit length at two random points to form three segments.

x	y	side1	side2	side3	triangle
0.6469028	0.7644140	0.6469028	0.1175112	0.2355860	FALSE
0.3942258	0.7438358	0.3942258	0.3496100	0.2561642	TRUE
0.6185018	0.8261657	0.6185018	0.2076639	0.1738343	FALSE
0.4768911	0.4227291	0.4227291	0.0541621	0.5231089	FALSE
0.1360972	0.4092877	0.1360972	0.2731905	0.5907123	FALSE
0.0673844	0.5396926	0.0673844	0.4723082	0.4603074	TRUE
0.1291526	0.9607224	0.1291526	0.8315698	0.0392776	FALSE
0.3931179	0.6535573	0.3931179	0.2604394	0.3464427	TRUE
0.0025827	0.5467153	0.0025827	0.5441326	0.4532847	FALSE
0.6202060	0.2660636	0.2660636	0.3541424	0.3797940	TRUE

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

```
mean(triangle)
```

```
## [1] 0.4
```

When we ask R to add the elements of a logical vector, it coerces the Boolean values to numbers: `TRUE` becomes 1 and `FALSE` becomes 0. Summing the numbers is then equivalent to counting how many times `TRUE` appeared in `triangle`. Taking the mean is equivalent to computing the proportion of `TRUE`s in `triangle`.

We would like, of course, to simulate breaking the unit length many times, and we would like to be able to choose easily—without a lot of fuss in coding—the number of repetitions. It makes sense, therefore, to write a simulation function that will do our work for us:

```
triangleSim <- function(reps = 10000 ) {
  cut1 <- runif(reps)
  cut2 <- runif(reps)
  triangle <- makesTriangle(cut1, cut2)
  cat("The proportion of triangles was ", mean(triangle), ".\n", sep = "")
}
```

Now that we think about it, it might also be nice to have the option to view a table of the results, along with the estimate of the probability:

```
triangleSim <- function(reps = 10000, table = FALSE ) {
  cut1 <- runif(reps)
  cut2 <- runif(reps)
  triangle <- makesTriangle(cut1, cut2)
  if ( table ) {
    cat("Here is a table of the results:\n")
    print(table(triangle))
    cat("\n")
  }
  cat("The proportion of triangles was ", mean(triangle), ".\n", sep = "")
}
```

Let's give our function a try, using the default of ten thousand repetitions and asking for a table of results:

```
triangleSim(table = TRUE)
```

```
## Here is a table of the results:
## triangle
## FALSE  TRUE
##  7466  2534
##
## The proportion of triangles was 0.2534.
```

As we would expect, our estimate of the probability of a triangle is pretty close to 1/4, the value that is known to experts in probability.

6.3.1 Setting a Seed

There are further possibilities for refining the simulation function. We know that it can be a good idea to set a seed for R's random-number generator. When you do this you still get random-looking results, but they will be the same results no matter how often you call the simulation function. In that way others who have access to your function can "reproduce" the results you got, thus assuring themselves that you weren't making anything up. Let's add a `seed` parameter to our simulation function:

```
triangleSim <- function(reps = 10000, table = FALSE, seed ) {
  set.seed(seed)
  cut1 <- runif(reps)
  cut2 <- runif(reps)
  triangle <- makesTriangle(cut1, cut2)
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(triangle))
    cat("\n")
  }
  cat("The proportion of triangles was ", mean(triangle), ".\n", sep = "")
}
```

This is all very well, but if for some reason you *desire* to have the function produce different results with every call, you have to come up with the seed yourself. It would be nice to have the option to call your function without having to provide a seed.

This is not difficult to accomplish. We'll set the default value for `seed` to be `NULL`. Then when the function begins we'll check the value of `seed`. If it is not `NULL`, then the user has provided a seed, and we'll use that. Otherwise, we'll let R grab pseudo-random numbers however it likes.

```
triangleSim <- function(reps = 10000, table = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  cut1 <- runif(reps)
  cut2 <- runif(reps)
  triangle <- makesTriangle(cut1, cut2)
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(triangle))
    cat("\n")
```

```
  }
  cat("The proportion of triangles was ", mean(triangle), ".\n", sep = "")
}
```

6.3.2 Program Development

Think about how we developed the `triangleSim()` function.

1. We began by writing simple code that got us a simulation for a few repetitions.
2. When we were sure how to make a simulation work, we *encapsulated* our code into a function that we could call to perform the simulation for us.
3. We though about features that we would like the function to have—user chooses number of repetitions, user chooses whether to see a table, etc.—and implemented these features one by one. This made the function more *general*, i.e., useful in a wider range of settings.

The above method of program development is called *encapsulation and generalization*. For small projects of the sort that we have on hand, it's a good way to go about writing a computer program.

6.3.3 Number of Repetitions

The Law of Large Numbers says that the more times you repeat the simulation, the better our estimate of the desired probability—or expected value—is liable to be. Let's use the Stick-Splitting Problem to investigate the effect of the choice of repetitions on the accuracy of our estimate of the probability of getting a triangle.

We will need to run the `triangleSim()` function several times, with a different choice for `reps` each time, keeping track of the estimates we obtain. In the process we won't need tables or other output to the console. Let's rewrite `triangleSim()` for maximum flexibility, making console output optional and using the invisible return discussed in Section 4.2.2.[2]

```
triangleSim <- function(reps = 10000, table = FALSE, seed = NULL,
                        verbose = TRUE) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  cut1 <- runif(reps)
  cut2 <- runif(reps)
  triangle <- makesTriangle(cut1, cut2)
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(triangle))
    cat("\n")
  }
  if ( verbose ) {
    cat("The proportion of triangles was ", mean(triangle), ".\n", sep = "")
  }
  invisible(mean(triangle))
}
```

Next, we make a vector of repetitions, and a vector to hold our estimates:

[2]The addition of the invisible return may be thought of as yet another step in the encapsulation-and-generalization process of program development!

```
reps <- c(100, 1000, 10000, 100000, 1000000)
estimates <- numeric(length(reps))
```

Next we loop through `reps`, performing a complete simulation for each repetition-number and storing our results:

```
for ( i in 1:length(reps)) {
  estimates[i] <- triangleSim(reps = reps[i], seed = 3030,
                              verbose = FALSE)
}
```

Let's have a look at our estimates:

```
names(estimates) <- reps
estimates
```

```
##      100     1000    10000    1e+05    1e+06
## 0.200000 0.245000 0.254700 0.251570 0.249779
```

The estimates are indeed getting closer to the theoretical value of 1/4.[3]

6.4 Example: Will They Connect?

Anna and Raj make a date for coffee tomorrow at the local Coffee Shop. After making the date, both of them forget the exact time they agreed to meet: they can only remember that it was to be sometime between 10 and 11am. Each person, independently of the other, randomly picks a time between 10 and 11 to arrive. If Anna arrives and Raj is not there, she will wait up to ten minutes for him, but will leave if he does not show within that time period. Raj is similarly disposed: he will wait ten minutes—but no more—for Anna. What is the chance that they meet?

The key to designing a simulation procedure is to realize that Anna and Raj will connect if and only if the difference between their arrival times is less than 10 minutes. It doesn't matter who arrived first, so long as the *difference* is less than 10. This means that we want to check on the *absolute value* of Anna's arrival time minus Raj's arrival time. The following code implements this idea:

```
meetupSim <- function(reps = 10000, table = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  anna <- runif(reps, 0, 60)
  raj <- runif(reps, 0, 60)
  connect <- (abs(anna - raj) < 10)
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(connect))
    cat("\n")
  }
```

[3]Statistical theory tells us that when we use Monte Carlo simulation to estimate a probability or an expected value, then the likely size of the error in the approximation is inversely proportional to the square root of the number of repetitions we use. From this it follows that to cut the likely size of the error in half you should increase the number of repetitions by a factor of 4. If you want the likely size to be one-third as big, increase repetitions by a factor of 9, and so on.

```
  cat("The proportion of times they met was ", mean(connect), ".\n", sep = "")
}
```

Let's try it out:

```
meetupSim(reps = 100000, table = TRUE, seed = 3939)
```

```
## Here is a table of the results:
##
## connect
## FALSE  TRUE
## 69781 30219
##
## The proportion of tims they met was 0.30219.
```

6.4.1 Vectorization vs. Looping

Whenever you perform a simulation you have the option to use a `for`-loop, running through the loop once for each repetition of the random process in question. For example, you could rewrite the meetup-simulation as follows:

```
meetupSim2 <- function(reps = 10000, table = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  #create an empty vector to hold the results:
  connect <- numeric(reps)
  # loop through:
  for ( i in 1:reps ) {
    # get one arrival time for anna:
    anna <- runif(1, 0, 60)
    # and one for raj:
    raj <- runif(1, 0, 60)
    #compute result and record in conncect:
    connect[i] <- (abs(anna - raj) < 10)
  }
  #the rest is the same as in meetupSim:
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(connect))
    cat("\n")
  }
  cat("The proportion of times they met was ", mean(connect), ".\n", sep = "")
}
```

Be aware, though, that when you use a loop in place of vectorization your routine is liable to run more slowly. We can measure the difference with the R-function `system.time()`. As an illustration, let's compare the running times for the vectorized and the looping versions of the meetup simulation.

First we get the time for the thousand repetitions of the meetup, using the original vectorized function:

```
system.time(meetupSim(reps = 10000, seed = 4040))
```

```
## The proportion of times they met was 0.3079.
##    user  system elapsed
##   0.002   0.001   0.003
```

Our concern is with the total elapsed time: a mere three-thousandths of a second!

Next we get the time for the same number of repetitions, using the looping implementation:

```
system.time(meetupSim2(reps = 10000, seed = 4040))
```

```
## The proportion of times they met was 0.2993.
##    user  system elapsed
##   0.056   0.007   0.065
```

Here the elapsed time is 0.065 seconds. This still seems pretty fast, but it's more than 20 times as long as for the vectorized simulation. For more complex simulations such a dramatic slowdown could pose serious practical problems.

When performance is an issue, prefer vectorization to looping as much as possible.

6.5 Example: the Appeals Court Paradox

The following example is discussed in (J.Nahin, 2008).

An appeals court generally consists of an odd number of justices, so the court will not be hampered with tie-votes and will thus always be able to render a decision for any case brought before it. Let's imagine an appeals court that has five members. Let us further imagine that for each case before the court, each of the justices, independently of the others, makes a decision for either one or the other of the two opposing parties. Assume also for the sake of simplicity that there is always one side that is "in the right". It follows that each judge is making a decision that is either correct or incorrect. The judges then report their decisions to one another, and the decision of the court is determined by majority vote. To be precise: whichever side gets three or more votes is the side that wins.

Although all of the members of the court are pretty sharp, they differ somewhat in their legal abilities:

- In any case that comes before her, Judge A has a 95% chance to decide correctly.
- Judge B has a 94% chance to judge correctly.
- Judges C and D each have a 90% chance to judge correctly.
- Judge E is the weak link on the court, with a mere 80% chance to judge correctly.

We are interested in estimating the probability that the majority opinion of the court will be correct.

Before we write a simulation in full, we should consider how we are going to simulate something that has a specified percentage chance of happening. For example, how would we simulate decisions made by Judge A?

One approach is to use `runif()` with a cutoff:

```
number <- runif(1) # random number between 0 and 1
number < 0.95
```

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

In repeated trials, a random real number between 0 and 1 will turn out to be less than 0.95 about 95% of the time.

Another—and more convenient—way is to use the function `rbinom()`. This function simulates the results of flipping a coin a number of times and counting how many heads one gets. The general form of a call to `rbinom()`is as follows:

```
rbinom(n, size, prob)
```

In this call:

- `size` is how many times you plan to flip the coin;
- `prob` is the chance on any flip that the coin will turn up heads;
- `n` is how many times you plan to repeat the process of flipping the coin `size` times (counting up the number of heads each time).

In order to simulate flipping a fair coin 100 times, you could ask for:

```
rbinom(n = 1, size = 100, prob = 0.5)
```

```
## [1] 52
```

It seems that we got 52 heads!

If you want to go through the above process twenty times instead of just doing it once, you could ask for:

```
rbinom(n = 20, size = 100, prob = 0.5)
```

```
##  [1] 52 49 45 57 53 60 46 53 50 54 59 47 42 45 41 56 47 57 56 49
```

Nothing in `rbinom()` says that it addresses only coin-flipping. In general it serves as a model for any situation in which:

- there is a fixed number of trials (e.g., flip of a coin, taking a free-throw shot, deciding about a court case);
- each trial has two possible outcomes, often termed Success or Failure (e.g., coins lands heads or tails, you make the shot or you don't, you decide the case correctly or you don't);
- the chance of the Success outcome is the same, on any trial (e.g., the coin always has the same chance of coming up heads, for *any* case Judge A has a 95% chance of being right);
- the outcome of any trial is independent of the outcome of all other trials (e.g., making the first free throw does not increase or decrease one's chances of making the next free throw, etc.)
- you are counting up the number of successes in your trials.

In such a situation the count of the number of successes is called a *binomial* random variable.[4] The `size` parameter in `rbinom()` gives the fixed number of trials. The `prob` parameter specifies the chance of success on each trial.

In this example we would like to simulate each judge deciding about each case. Suppose for example, that there are twenty cases. We would like to know, for each judge and each case, whether or not the judge was correct

Suppose we were to watch Judge A making 20 decisions, recording each time whether she was correct or not. In order to simulate the results with R, we could ask for:

[4] "Binomial" is borrowed from a Greek words whose literal meaning is "two names"—corresponding to the two possible outcomes of a trial.

```
rbinom(n = 20, size = 1, prob = 0.95)
```

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

In the above results, each 1 stands for a correct decision and the 0 stands for an incorrect decision.

Note that the following call would be incorrect:

```
rbinom(n = 1, size = 20, prob = 0.95)
```

This would give us a *count* of the number of times Judge A was correct, but it would not tell us which of the cases she judged correctly and which she judged incorrectly.

Assuming that the court hears 20 cases, we would like to simulate the decision of each judge in each case. We could store the results in variables, as follows:

```
a <- rbinom(n = 20, size = 1, prob = 0.95)  # Judge A
b <- rbinom(n = 20, size = 1, prob = 0.94)  # Judge B
c <- rbinom(n = 20, size = 1, prob = 0.90)  # Judge C
d <- rbinom(n = 20, size = 1, prob = 0.90)  # Judge D
e <- rbinom(n = 20, size = 1, prob = 0.80)  # Judge E
```

Now we come to the key idea in coding our simulation: in order to determine the total number of correct votes in each case, all we need to do is *add* the five vectors we created above:

```
correctVotes <- a + b + c + d + e
```

The court will decide correctly when the number of votes for the correct option is at least 3. The following logical vector records this:

```
courtCorrect <- (correctVotes >= 3)
```

Table 6.2 shows some possible results:

Note that all of the court's decisions were correct, even though in a couple of cases the correct decisions were barely in the majority.

We now have the basic idea of the simulation. In order to estimate the probability of a correct decision, we simply need to recast our idea in the form of a function that permits us to simulate a very large number of imaginary court cases and that will report the results to us. Here is the code for such a function.

```
courtSim <- function(reps = 10000,
                     seed = NULL,
                     table = FALSE,
                     probs = c(0.95, 0.94, 0.90, 0.90, 0.80)) {

  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # get the probabilities
  aProb <- probs[1]
  bProb <- probs[2]
```

Table 6.2: Results of 20 simulated appeals court decisions.

A	B	C	D	E	Number Correct	Court Correct
1	1	1	0	0	3	TRUE
1	1	1	0	1	4	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	1	5	TRUE
1	1	1	0	1	4	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	0	4	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	1	5	TRUE
1	1	1	0	1	4	TRUE
1	1	0	1	0	3	TRUE
1	1	1	1	1	5	TRUE
1	1	1	0	1	4	TRUE
0	1	1	1	1	4	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	1	5	TRUE
1	1	1	1	0	4	TRUE
1	1	0	1	1	4	TRUE

```
  cProb <- probs[3]
  dProb <- probs[4]
  eProb <- probs[5]

  # simulate decisions of each judge:
  a <- rbinom(n = reps, size = 1, prob = aProb)
  b <- rbinom(n = reps, size = 1, prob = bProb)
  c <- rbinom(n = reps, size = 1, prob = cProb)
  d <- rbinom(n = reps, size = 1, prob = dProb)
  e <- rbinom(n = reps, size = 1, prob = eProb)

  # count the number of correct votes in each case:
  correctVotes <- a + b + c + d + e

  # determine whether court decided correctly, in each case:
  courtCorrect <- (correctVotes >= 3)

  # record results
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(courtCorrect))
    cat("\n")
  }
  cat("The proportion of times the court was correct was ",
      mean(courtCorrect), ".\n", sep = "")
}
```

Let's now estimate the probability of the court rendering a correct verdict, using one hundred thousand

simulated cases:

```
courtSim(reps = 100000, seed = 3838, table = TRUE)
```

```
## Here is a table of the results:
##
## courtCorrect
## FALSE  TRUE
##   787 99213
##
## The proportion of times the court was correct was 0.99213.
```

The court seems to be doing quite well! But of course this is not very surprising: after all, most of the judges make the correct decision almost all the time. It is interesting, though, that the chance of the *full court* rendering a correct verdict is higher than the chance of any individual judge to be correct. There appears to be a benefit to the voting procedure.

But things get even more interesting if we imagine that the judges recognize, after some time, that Judge E simply isn't as sharp as Judge A, and that they pressure him into voting whichever way Judge A votes. "Surely", they reason, "since Judge A is so good this new policy will increase our chance to hand down a correct verdict!"

We will simulate to see if the judges' reasoning is correct. All we need to do is to count Judge A's vote twice in the sum, and not count Judge E's vote at all:

```
correctVotes <- 2*a + b + c + d
```

Here is the new simulation-function:

```
courtSim2 <- function(reps = 10000,
                      seed = NULL,
                      table = FALSE,
                      probs = c(0.95, 0.94, 0.90, 0.90, 0.80)) {

  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # get the probabilities
  aProb <- probs[1]
  bProb <- probs[2]
  cProb <- probs[3]
  dProb <- probs[4]
  eProb <- probs[5]

  # simulate decisions (this time, no need for Judge E) :
  a <- rbinom(n = reps, size = 1, prob = aProb)
  b <- rbinom(n = reps, size = 1, prob = bProb)
  c <- rbinom(n = reps, size = 1, prob = cProb)
  d <- rbinom(n = reps, size = 1, prob = dProb)

  # count rhe number of correct votes in each case:
  correctVotes <- 2*a + b + c + d
```

```
  # determine whether court decided correctly, in each case:
  courtCorrect <- (correctVotes >= 3)

  # record results
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(courtCorrect))
    cat("\n")
  }
  cat("The proportion of times the court was correct was ",
      mean(courtCorrect), ".\n", sep = "")
}
```

Let's try it out:

```
courtSim2(reps = 100000, seed = 3838, table = TRUE)
```

```
## Here is a table of the results:
##
## courtCorrect
## FALSE  TRUE
##  1221 98779
##
## The proportion of times the court was correct was 0.98779.
```

Hey! That's a *lower* chance of success than before! The difference is small but significant: compelling the weakest judge to vote with the strongest judge actually *decreased* the court's chance of rendering a correct verdict overall. This circumstance is sometimes called the Appeals Court Paradox, but it occurs in many other practical situations. More often than you might think, the benefit of independent voting can outweigh the advantage associated with relying solely on a small number of "experts."

6.6 Example: How Many Numbers Needed?

Let us pause for a review.

So far our examples have involved estimating a probability: the chance for a particular event to occur. We have accomplished this with Monte Carlo simulation:

- We repeated the chance process a very large number of times
- On each repetition, we recorded whether or not the event occurred.
- After we finished all of the repetitions, we computed the proportion of those repetitions in which the event *did* occur.
- This proportion served as our estimate of the probability for the event to occur.
- We know that the more repetitions we arrange for, the closer our estimate is liable to be to the actual probability for the event to occur.

Let us now switch gears and take up the task of estimating the *expected value* of a random variable.

Recall that a random variable is a number whose value is the result of some chance process. The expected value of the random variable is the *average*—the mean—of its values over many, many repetitions of that chance process.

Let us consider the following chance process:

- Pick a real number x_1 between 0 and 1 at random.
- Pick another real number x_2 at random (also between 0 and 1). Add it to x_1.
- If the sum $x_1 + x_2$ exceeds 1, then stop.
- Otherwise, pick a third real number x_3, and add it to the sum of the previous two.
- If the sum $x_1 + x_2 + x_3$ exceeds 1, then stop.
- Otherwise, pick a fourth real number …

The idea is to keep on going in this way until the sum of the numbers you have picked exceeds 1.

The random variable we are interested in is the *number* of numbers we have to pick to make the sum of those numbers exceed 1. Let us call this random variable X. As you can see, this number could be 2, or 3, or 4, or even more depending on how small the random numbers you pick happen to be.

In order to fully grasp what is going on, it will be helpful to go through the process of computing X. The following function will do this for us:

```
numberNeeded <- function(verbose = FALSE) {
  mySum <- 0
  count <- 0
  while( mySum < 1 ) {
    number <- runif(1)
    mySum <- mySum + number
    count <- count + 1
    if ( verbose ) {
      cat("Just now picked ", number, ".\n", sep = "")
      cat(count, " number(s) picked so far.\n", sep = "")
      cat("Their sum is ", mySum, ".\n\n", sep = "")
    }
  }
  count
}
```

Call the function a few times, like this:

```
numberNeeded(verbose = TRUE)
```

```
## Just now picked 0.2869393.
## 1 number(s) picked so far.
## Their sum is 0.2869393.
##
## Just now picked 0.7092682.
## 2 number(s) picked so far.
## Their sum is 0.9962075.
##
## Just now picked 0.7524099.
## 3 number(s) picked so far.
## Their sum is 1.748617.
```

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

As you can see, it's quite common to need 2 or 3 numbers to make the sum exceed 1, but sometimes, you need 4, 5 or even more numbers.

Since we are interested in the expected value of X, we should repeat the process a large number of times and compute the mean of the X-values that we find. The following code repeats the process 1000 times:

```
needed <- numeric(1000)
for (i in 1:1000 ) {
  needed[i] <- numberNeeded()
}
cat("The expected number needed is about ",
    mean(needed), ".\n", sep = "")
```

```
## The expected number needed is about 2.697.
```

Of course the expected value indicates what X will be *on average*. By itself it doesn't tell us about the *variability* in X—how much X tends to "hop around" from one repetition to another. In order to get a sense of the variability of X, we can make a table of the results of our 1000 repetitions:

```
table(needed)
```

```
## needed
##   2   3   4   5   6   7
## 528 303 123  37   8   1
```

Sure enough, we usually got 2 or 3, but higher values are certainly possible.

For the most part we are less interested in the *number* of times we got each possible value than we are in the *proportion* of times we got each possible value. For this all we need is `prop.table()`:

```
prop.table(table(needed))
```

```
## needed
##     2     3     4     5     6     7
## 0.528 0.303 0.123 0.037 0.008 0.001
```

The proportions serve as estimates of the *probability* of getting each of the possible values of X. Such a table of proportions gives us a sense of the *distribution* of X—the chances for X to assume each of its possible values.[5]

Now that we have the basic idea of how to accomplish the simulation, it remains to package up our code into a nice function that will enable a user to set the number of repetitions, set a seed, and choose reporting options. Along the way we will generalize a bit, allowing the user to set a target sum other than 1. We'll also return our estimate of the expected number needed invisibly, so that the function can be used in the context of larger-scale investigations (see the exercises from this Chapter).

```
numberNeededSim <- function(target = 1, reps = 1000,
                            seed = NULL, report = FALSE) {

  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # define helper function
  numberNeeded <- function(target) {
    mySum <- 0
    count <- 0
```

[5] Of course not all of the possible values for X appear on the table: the only values appearing are those that happened to occur in our simulation. The values that did not occur probably have a very small (but nonzero) chance of occurring.

```
    while( mySum < target ) {
      number <- runif(1)
     mySum <- mySum + number
      count <- count + 1
    }
    count
  }

  #perform simulation
  needed <- numeric(reps)
  for (i in 1:reps ) {
    needed[i] <- numberNeeded(target)
  }

  # report results (if desired)
  if ( report ) {
    print(prop.table(table(needed)))
    cat("\n")
    cat("The expected number needed is about ",
        mean(needed), ".\n", sep = "")
  }
  # return estimate of the expected number needed:
  invisible(mean(needed))
}
```

Let's test the function with ten thousand repetitions:

```
numberNeededSim(target = 1, reps = 10000, seed = 4848, report = TRUE)
```

```
## needed
##      2      3      4      5      6      7      8
## 0.4937 0.3350 0.1313 0.0317 0.0071 0.0010 0.0002
##
## The expected number needed is about 2.7273.
```

Does the estimate of the expected number of numbers look a bit familiar? Mathematicians have proven that the exact expected value—what you would get for an average if you could do more and more repetitions, without limit—is the famous number e:

$$\mathrm{e} \approx 2.71828\ldots$$

In the exercises you will be asked to investigate the expected value when the target is a number other than 1.

6.7 Example: Dental Floss

If you are like me, you don't like to run out of an item that you commonly use, so you always try to have a spare source for the item on hand. Take dental floss, for example. It's difficult to know when the little box is almost out of floss. Hence I try to keep *two* boxes on hand in my bathroom drawer: the one I'm using, and a spare. The problem is that I can never tell which one is "the one I'm using" and which one is the

spare. In fact on any given day I'm equally likely to pull a length of floss out of either one of the boxes.[6]

Let's make the following assumptions:

- A box of floss starts out with 150 feet of floss.
- I start with two fresh boxes.
- Every time I floss, each box has a 50% chance of being the box I pull from.
- When I pull out floss, the length I choose to pull is a random number between 1 and 2 feet. If the box has less than than the amount I choose to pull, then:
  - If the remaining length is more than 1 foot, I pull it all out and use it. (Next time I'll have to use the other box.)
  - If the remaining length is less than 1 foot, I also pull it all out, but I don't use it because it's not long enough. (I'll have to move on to the other box.)

There are several interesting questions we could ask concerning this situation. In this example, let's focus on finding the expected value of X, the amount of floss left in the *other* box when I discover the one of the boxes is exhausted.

I would also like to get some idea of the distribution of X—the chance for it to assume each of its possible values. Unlike the previous "Number of Numbers" example, however, X is what we call a *continuous* random variable: it can assume any of the values in a *range of real numbers*. After all, the amount left in the "other" box could be anything from 0 to 150 feet: it could be 3.5633 feet or 5.21887 feet, or any real number at all, so long as it's between 0 and 150. Although any real number value—3.5633 feet, for instance—is *possible*, it seems that the chance of X being *exactly* 3.5633 is vanishingly small, and the same goes for any of the possible real numbers value for X. Hence it seems that the distribution of X cannot be described well with a table of the values that we actually get in a simulation: indeed, we are highly unlikely to get any value more than once in the course of our simulation.

One way to visualize the distribution of a continuous random variable is to make a *density plot* of a large, representative sample of its values. Let's practice making a density plot for a random variable that R provides for us: a so-called *normal* random variable. Try this code:

```
x <- rnorm(100, mean = 70, sd = 3)
```

We have just asked R to produce 100 numbers chosen randomly from a *normal* distribution. The `mean` parameter sets what the values should be "on average", and the `sd` parameter indicates how much they should typically differ from that average value. The bigger the `sd`, the bigger the difference between the mean and value actually obtained is liable to be.

Let's get a look at the first ten values in the vector `x`:

```
x[1:10]
```

```
##  [1] 71.13092 70.90465 66.70593 66.60878 61.61040 72.16172 72.81736
##  [8] 69.31187 75.27739 70.35210
```

Sure enough, the numbers are typically around the mean of 70, give or take 3 units or so.

We can use the **ggplot2** package to produce a density curve from all 100 values in the vector `x`. (The resulting plot appears as Figure 6.2.)

```
library(ggplot2)
```

[6] A scenario similar to this one is discussed in (J.Nahin, 2008), see p. 52.

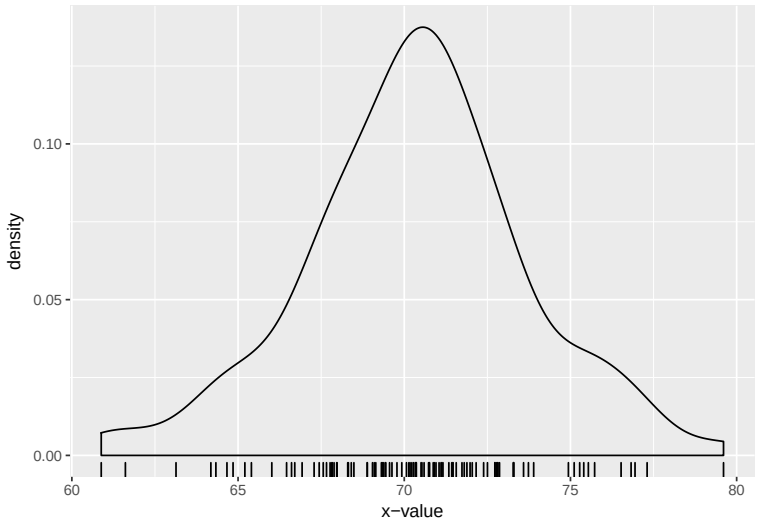


Figure 6.2: Density plot of 100 random values from a normal distribution with mean 70 and standard deviation 3.

```
p <- ggplot(mapping = aes(x = x)) + geom_density() + geom_rug() +
  labs(x = "x-value")
p
```

The `geom_density()` part of the plotting-command produces the curve. The `geom_rug()` part produces tick-marks along the horizontal axis, each of which is located at a particular value in the vector `x`. You can see that the curve is higher where values are crowded together, and lower where the values are spread apart. The scale on the y-axis has been chosen carefully so that the area under the density curve between any two numbers a and b on the axis gives an estimate of the chance that the normal random variable would produce a value between a and b. In this way the density curve gives us a visual sense of the distribution of the random variable: it seems rather likely to be between 67 and 73 (most of the area under the curve occurs there) and rather unlikely to be greater than 76, and so on.

We are now ready to simulate the dental-floss scenario. Since we have worked through the encapsulation-and-generalization development process a number of times by now, we will simply show a final-form implementation of a simulation function:

```
flossSim <- function(len = 150, min = 1,
                     max = 2, reps = 10000,
                     graph = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # helper function to pick whihc box gets used:
  whichOne <- function(x) {
    if (x < 0.5 ) {
      return("a")
    } else return("b")
  }

  # vector to contains our results:
```

```
leftover <- numeric(reps)

# now the simulation.  Each time we loop through, we
# start with two fresh boxes and use them until one of
# them runs out.
for (i in 1:reps ) {
  # set the initial lengh of the two fresh boxes:
  a <- b <- len

  # at the beignning, both can be used:
  bothOK <- TRUE

  # start flossing
  while ( bothOK ) {
    # determine which box is picked
    # < 0.5 is a; >= 0.5 is b;
    boxPicked <- whichOne(runif(1))

    # if we choose box a, attempt to use floss from it
    if ( boxPicked == "a" ) {
      if ( a < 1 ) {
        leftover[i] <- b
        bothOK <- FALSE
      } else {
        useAmount <- min(runif(1, min, max), a)
        a <- a - useAmount
        if (abs(a) < 10^(-4)) {
          leftover[i] <- b
          bothOK <- FALSE
        }
      }
    }

    # use floss from b (if we choose it)
    if ( boxPicked == "b" ) {
      if ( b < 1 ) {
        leftover[i] <- a
        bothOK <- FALSE
      } else {
        useAmount <- min(runif(1, min, max), b)
        b <- b - useAmount
        if (abs(b) < 10^(-4)) {
          leftover[i] <- a
          bothOK <- FALSE
        }
      }
    }

  }  # end while  loop
}    # end for loop

# report results:
```

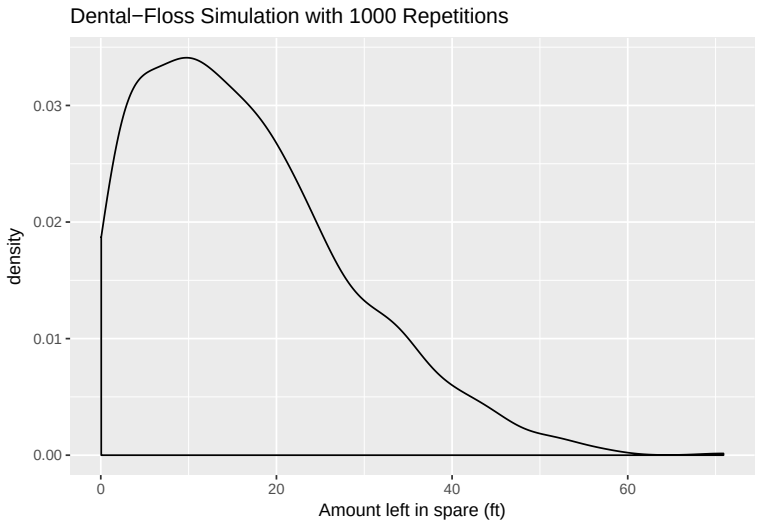


Figure 6.3: Density plot of the results of a dental-floss simulation.

```
  if ( graph ) {
    plotTitle <- paste0("Dental-Floss Simulation with ", reps, " Repetitions")
    p <- ggplot(mapping = aes(leftover)) + geom_density() +
      labs(x = "Amount left in spare (ft)",
           title = plotTitle)
    # add a rug of individual points, provided there are not too many:
    if ( reps <= 100 ) {
      p <- p + geom_rug()
    }
  }
  print(p)
  cat("The average amount left in the spare is:  ",
      mean(leftover), ".", sep = "")
}      # end flossSim
```

Let's give it a try with one thousand repetitions.

```
flossSim(reps = 1000, graph = TRUE, seed = 3535)
```

```
## The average amount left in the spare is:  16.55715.
```

A density plot of the results appears as Figure 6.3. The expected value of the amount left in the remaining box is about 16.3 feet, but you can see from the plot that there is quite a bit of variability: it's not all that unlikely to have 40 or more feet left!

6.8 Example: The Drunken Turtle

Let us return to the scenario of the Drunken Turtle that was introduced in Section 5.4.5. At each step the turtle would turn by a random angle—any angle between 0 and 360 degrees with equal likelihood—and then move forward by a fixed number of units. We wondered whether the turtle would simply wander away from its starting point or perhaps eventually loop back close to its starting point.

In this section we will construct a simulation that sheds light on our question. We will imagine that the turtle starts at the origin on the xy-plane and takes a large but fixed number of steps—one thousand steps, say—each of a fixed length of one unit, but each of which is preceded by a drunken turn through a random angle. We will concern ourselves with the random variable X, defined as follows:

> X = the number of times in the turtle returns to within 1/2 of a unit of the origin, during its first 1000 steps.

(Apparently we consider a "close return" to be anything within half a unit of the origin.)

In particular, we would like to know the distribution of X: what is the chance that the turtle does not make a close return during the first thousand steps? What's the chance of making only one close return? Two close returns? And so on … . We would also like to estimate the expected value of X: the average number of close returns in the first 1000 steps.

We will next consider how to code a simulation. Obviously we will need a way to record the turtle's position at each step. In order to accomplish this we will have to apply the unit-circle trigonometry you learned in high school.

Instead of measuring angles in degrees, we will work in radians: by default R does trigonometry that way. Recall that 360 degrees is 2π radians; hence the way to determine the angle of a turn by the drunken turtle is with the following call to `runif()`:

```
runif(1, min = 0, max = 2*pi)
```

```
## [1] 4.146176
```

If we want a thousand drunken turn-angles, then we can get them all in a vector, as follows:

```
runif(1000, min = 0, max = 2*pi)
```

So much for the turning angles. But in order to decide at each step whether the turtle has made a close return, we have to know the coordinates of the turtle after each step. This is where unit-circle trigonometry comes in handy. Recall that if a right triangle has a hypotenuse of length 1 (the length of a turtle step) and if one of the angles of the triangle is α, then the side adjacent to α is $\cos(\alpha)$ and the side opposite is $\sin(\alpha)$. The situation is illustrated in Figure 6.4.

From this principle it follows that if the turtle turns a random angle of a_1 radians for its first step, then after it takes the one-unit step it will be at the point $(\cos(a_1), \sin(a_2))$.

For the second step, the turtle begins by turning another random angle, say a_2 radians. It then takes another one-unit step. This second step forms the hypotenuse of a second right triangle (see Figure 6.5). The horizontal and vertical sides of this right triangle must be $\cos(a_2)$ and $\sin(a_2)$ respectively. When the turtle complete the second step, it has added these values to its previous x and y coordinates, so that it is now at the point:

$$(\cos(a_1) + \cos(a_2), \sin(a_1) + \sin(a_2)).$$

We now see the idea: in order to record the position of the turtle at the end of each step, we simply need to:

- take the cosine of all the turning-angles;
- take the sine of all the turning-angles;
- sum the cosines to get the x-coordinate;
- sum the sines to get the y-coordinate.

Let's put this idea into practice, for just the first ten steps. First, get the turning angles, the sines and the cosines:

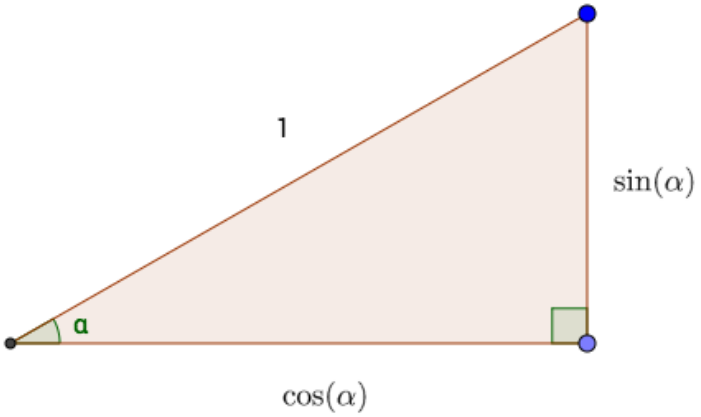


Figure 6.4: Sides of a right triangle with hypotenuse 1, in terms of the angle shown.

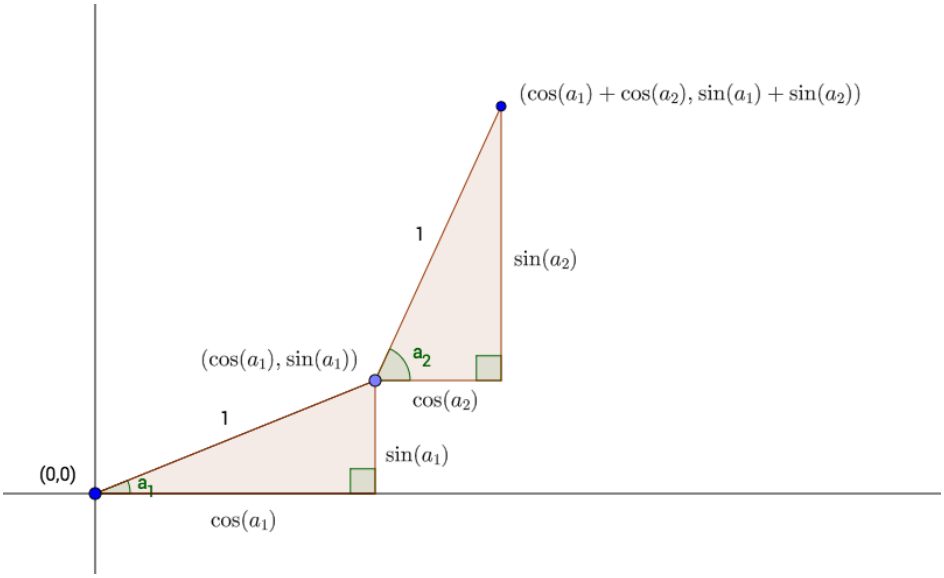


Figure 6.5: The first two steps by the Drunken Turtle. Turtle turns by a_1 radians for Step One, then by a_2 radians for Step Two.

```
angle <- runif(10, 0 , 2*pi)
xSteps <- cos(angle)
ySteps <- sin(angle)
```

Next, we need to add the `xSteps` together. If we do a sum, we would get the x-coordinate after the tenth step:

```
sum(xSteps)
```

```
## [1] -1.370258
```

(Apparently some of the cosines were negative, meaning that some of the turning angles were between 90 and 270 degrees.)

We need the x-coordinates after all of the previous steps, as well. This can be accomplished by R's `cumsum()` function. You may think of `cumsum` as short for "cumulative sum". Given a vector `vec` of numbers, `cumsum(vec)` returns a vector of the same length, where for every index `i`, `cumsum(vec)[i]` is `sum(vec[1:i])`. This is made clear in the following example:

```
vec <- c(3, 2, -1, 4, -2)
cumsum(vec)
```

```
## [1] 3 5 4 8 6
```

Look at the table below.

`vec`	cumulative sum	`cumsum(vec)`
3	3	3
2	$3+2$	5
-1	$3+2+-1$	4
4	$3+2+-1+4$	8
-2	$3+2+-1+4+-2$	6

See how it works?

Returning to `xSteps`, we see that we can get the x-coordinates of the first ten steps like this:

```
x <- cumsum(xSteps)
x
```

```
##  [1] -0.6034165 -1.3905805 -2.1259607 -3.1154380 -2.4593130 -1.5476110
##  [7] -0.8591924 -1.6420445 -0.6421761 -1.3702583
```

Similarly we can get the first ten y-coordinates:

```
y <- cumsum(ySteps)
```

Using the distance formula, we can compute the distance of each point from the origin as follows:

```
dist <- sqrt(x^2 + y^2)
```

Table 6.4: Record of the turtle's first ten steps. It has not yet made a close return.

angle	xSteps	ySteps	x	y	dist	closeReturn
4.0646104	-0.6034165	-0.7974262	-0.6034165	-0.7974262	1.000000	FALSE
2.4769935	-0.7871641	0.6167437	-1.3905805	-0.1806826	1.402270	FALSE
3.8861615	-0.7353801	-0.6776548	-2.1259607	-0.8583374	2.292695	FALSE
2.9963954	-0.9894774	0.1446876	-3.1154380	-0.7136497	3.196131	FALSE
0.8551238	0.6561251	0.7546522	-2.4593130	0.0410024	2.459655	FALSE
0.4233886	0.9117020	0.4108522	-1.5476110	0.4518546	1.612226	FALSE
0.8114898	0.6884186	0.7253136	-0.8591924	1.1771682	1.457373	FALSE
2.4700328	-0.7828521	0.6222079	-1.6420445	1.7993761	2.435993	FALSE
0.0162276	0.9998683	0.0162269	-0.6421761	1.8156030	1.925826	FALSE
3.8968689	-0.7280822	-0.6854899	-1.3702583	1.1301131	1.776165	FALSE

The following logical vector will then record, at each each step, whether or not the turtle has made a close return:

```
closeReturn <- (dist < 0.5)
```

The results are shown in Table 6.4.

It remains only to package our code into a simulation function. We will allow the user to specify the fixed initial number of steps (it does not have to be 1000), and also to determine what distance counts as a "close return."

```
drunkenSim <- function(steps = 1000, reps = 10000, close = 0.5,
                       seed = NULL, table = FALSE) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # set up returns vector to store the number of
  # close returns in each repetition:
  returns <- numeric(reps)

  for (i in 1:reps) {
  angle <- runif(steps, 0 , 2*pi)
  xSteps <- cos(angle)
  ySteps <- sin(angle)

  x <- cumsum(xSteps)
  y <- cumsum(ySteps)

  dist <- sqrt(x^2 + y^2)
  closeReturn <- (dist < 0.5)
  returns[i] <- sum(closeReturn)
  }

  if ( table ) {
    cat("Here is a table of the number of close returns:\n\n")
    tab <- prop.table(table(returns))
    print(tab)
```

```
    cat("\n")
  }
  cat("The average number of close returns was:  ",
      mean(returns), ".", sep = "")
}
```

Let's try it out:

```
drunkenSim(steps = 1000, reps = 10000,
           close = 0.5, seed = 3535, table = TRUE)
```

```
## Here is a table of the number of close returns:
##
## returns
##      0      1      2      3      4      5      6      7      8      9
## 0.3881 0.2290 0.1517 0.0908 0.0572 0.0335 0.0227 0.0107 0.0064 0.0048
##     10     11     12     13     14     15     16     17
## 0.0018 0.0012 0.0008 0.0007 0.0003 0.0001 0.0001 0.0001
##
## The average number of close returns was:  1.5655.
```

Notice that 38.81% of the time the turtle did not make a close return at all during the first thousand steps. On the other hand it is possible, though not at all likely, for the turtle to make quite a few close returns.[7]

It is interesting to look specifically at the distances from the origin. The following function draws a graph of the distance from the origin for the specified number of turtle steps:

```
drunkenSim2 <- function(steps = 1000, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  angle <- runif(steps, 0 , 2*pi)
  xSteps <- cos(angle)
  ySteps <- sin(angle)
  x <- cumsum(xSteps)
  y <- cumsum(ySteps)
  dist <- sqrt(x^2 + y^2)
  plotTitle <- paste0("First ", steps, " Steps of the Drunken Turtle")
  p <- ggplot(mapping = aes(x = 1:steps, y=dist)) + geom_line() +
    labs(x = "Step", y = "Distance From Origin",
         title = plotTitle)
  print(p)
}
```

Let's try it out! The results appear in Figure 6.6

```
drunkenSim2(steps = 1000, seed = 2525)
```

It appears that although the turtle might have returned close to the origin a couple of times, for the most part it seems to be wandering further and further away. In the mathematical field known as *stochastic*

[7] Note that these proportions depend heavily on how we define "close". If the cut-off had been lower than 1/2, then we would have seen fewer close returns.

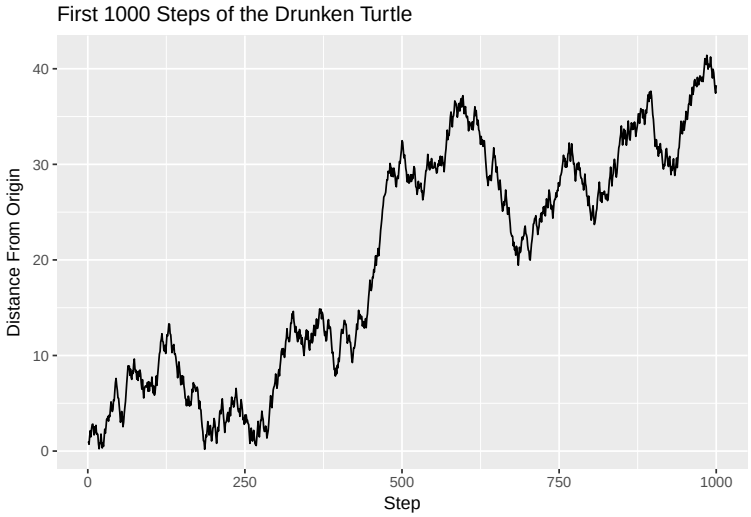


Figure 6.6: Line plot of the Turtle's distance from the origin, as a function of how many steps the turtle has taken.

processes, it has been shown that if the turtle is allowed to go on forever then it will certainly make a close return *infinitely many times*. These returns, though, will tend to be spaced further and further apart as time goes on, and our graph suggests that this is so.

Glossary

Probability of an Event The long-run proportion of times that the event will occur if the underlying random process is repeated many, many times.

Random Variable A number whose value depends on the outcome of a chance process.

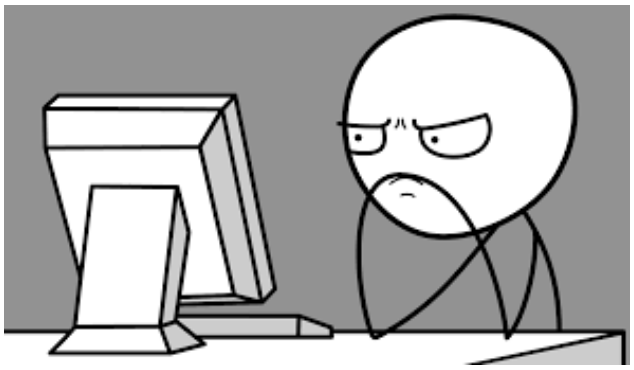
Expected Value of a Random Variable The long-run average of the values of a random variable, where the underlying chance process is repeated many, many times.

Continuous Random Variable A random variable that can assume any value in a range of real numbers.

Distribution of a Random Variable A statement of the probabilities for a random variable to assume its various possible values.

Encapsulation and Generalization A method for developing a computer program according to which the programmer first designs a basic procedure, then encapsulates it in one or more functions, and then progressively generalizes these functions until the program possesses all desired features.

Exercises



1. Consider the following two games.

   - In Game A you flip a fair coin. If the coin comes up Heads you get two dollars, whereas if it comes up Tails you get one dollar.
   - In Game B you roll a fair die. If the six-spot comes up, you win twenty-five dollars. If you get 2, 3, 4, or 5, nothing happens. If the one-spot comes up, you lose fifteen dollars.

   In this exercise you need to answer two question:

   1. If you could choose to play either Game A or Game B *just once*, which game would you prefer to play, and why?
   2. Suppose that you can choose either to play Game A ten thousand times, or Game B ten thousand times. Which choice would you prefer, and why?

   Answers to the first question will vary from person to person, depending on circumstances and personal taste. On the other hand, astute consideration of expected values leads most people to answer the second question in the same way. Therefore, before you answer the question please write two simulation functions: one called `GameA` that provides an estimate of the expected value of the winnings from a play of Game A, and another called `GameB` that estimates the expected value of the winnings from a play of Game B. The functions should have two parameters:

   - a `reps` parameter to permit the user to choose the number of repetitions of the game;
   - a `seed` parameter to permit the user to choose a starting seed.

   Use both functions with fifty thousand reps, in order to obtain estimates for the expected amount of winnings in a single play of each game. Use a simple starting seed for each function. Report the seeds you used, along with the estimated expected values for each game. Use your results to decide which game is the better choice to play ten thusand times.

   Note: Please keep in mind that the fifty-thousand figure is just the number of times you repeat your simulations in order to estimate the expected value of a *single play* of one of the games. It's **not** how many times you actually get to play either game—that's ten thousand.

2. Reconsider the meeting of Anna and Raj. Suppose that they still plan to meet during the same one-hour period, but that Anna is willing to wait a minutes and Raj is willing to wait r minutes, where a and r could differ. Modify the `meetupSim()` function so that it accepts two additional parameters:

   - `a`: the time Anna is willing to wait;
   - `r`: the time Raj is willing to wait;

   Apply the function with ten thousand repetitions (and a simple seed that you report) in order to estimate the probability of a meeting, assuming that Anna is willing to wait 13 minutes and Raj is willing to wait 7 minutes.

Hint: Suppose that Anna arrvies at time $Anna$ and Raj arrives at times raj. We want a simple mathematical expression that gives the conditions for Anna and Raj to connect. In the example in the text, this was:

$$\mid anna - raj \mid < 10$$

because both Anna and Raj were willing to wait the same amount of time: ten minutes. Now we have to think a bit more carefully when Anna waits a minutes and Raj waits r minutes, where a and r may differ. You might start by observing that we must have

$$anna - r \leq raj,$$

for if not then Raj would arrive and leave before Anna gets there. Also, we would have to have

$$raj \leq anna + a,$$

for if not then Anna would arrive and leave before Raj gets there. But if we have *both* of those conditions:

$$anna - r \leq raj \text{ and } raj \leq anna + a$$

then they must connect, since neither person will arrive too much before the other.

3. Refer back to the `numberNeededSim()` function. Write a program that computes the estimated expected number needed, for the following targets:

```
targets <- seq(0.05, 0.95, by = 0.05)
```

Each estimate should be based on ten thousand repetitions. Start with a simple seed that you report. Report the estimated expected values for all of the required targets. Compare these estimates with `exp(targets)` (the number e raised to the power of each of the targets.) Formulate a conjecture about the value of the expected number of numbers needed, when the target is a real number between 0 and 1.

Hint: Recall that `numberNeededSim()` function returns its estimate of the expected number needed. Hence you could use it inside a `for`-loop that iterates through the elements of the `targets` vector above, something like this:

```
targets <- seq(0.05, 0.95, by = 0.05)
n <- length(targets)
results <- numeric(n)
set.seed(3030) # or some other simple seed that you like
for ( i in 1:n ) {
  # use numberNeededSim() with:
  #    * 10000 reps
  #    * seed left at the default NULL (you provided one already)
  #    * target set to targets[i]
  #    * report left at FALSE (no need to have R talk to you at each step)
  # make sure to store the estimate in results[i]
}
# after the loop, you can compare results with exp(targets)
```

4. (*) A pipe-smoker has two boxes of matches in his pocket. Both boxes contain 40 matches. Every time he lights his pipe, the smoker reaches into his pocket and randomly picks one of the boxes, pulling a match from the box. Eventually one of the boxes runs out. What is the expected value of the number of matches remaining in the other box at that time? Write a simulation function to estimate the answer. Your function should have at least a `reps` and a `seed` parameter. Apply the function with ten thousand repetitions—and with a simple seed that you report—in order to estimate the expected value.

5. (*) A gambler starts with 10 dollars, and repeatedly plays a game. If she wins the game she gets one dollar and if she loses the game she loses one dollar. Her chance of winning each game is a fixed number that is less than 0.50. Write a function to simulate the gambler repeatedly playing the game until her money runs out. The function should keep track of how much money she has left after each play, so that it can produce a line graph (similar to the one in the section on the Drunken Turtle) of the money left after each play. Write the function so that it has

   - a `start` parameter for the initial amount of money the gambler has;
   - a `seed` parameter‘;
   - a `p` parameter for the chance of winning, so that the user can enter any chance less than or equal to 0.5. (The function should stop the user if the user enters more than 0.5.)

   Apply the function with a simple seed that you report, a starting amount of ten dollars, and a chance of winning equal to 0.45.

6. (*) An absent-minded professor walks to and from campus every day. Prior to every commute—either from home to office or the reverse—she checks the weather, taking an umbrella with her if it is raining. The problem is that she *never* takes an umbrella when the weather is dry, so even though she owns quite a few umbrellas the time comes eventually when it is raining but all of her umbrellas are at her destination, so that she will have to put up with a wet commute.

   Suppose that there are x umbrellas at her house and y umbrellas at her office, and that she begins at home. Suppose further that the chance of rain on any commute is p, a number between 0 and 1.

   Write a simulation function to estimate the expected number of dry commutes that she will enjoy before her first wet commute. The function should have five parameters:

   - `x`: the initial number of umbrellas at home (default 1);
   - `y`: the initial number of umbrellas at the office (default 1);
   - `p`: the probability of rain at any commute (default 0.5);
   - `seed`: an seed for the randomizers (default `NULL`).
   - `reps`: the number of times to simulate commuting until wet.

   Apply the function with `x` and `y` set at three, for the values 0.1, 0.5 and 0.9 for `p`. Use 10,000 repetitions for each simulation, with seeds that you report. Which value of `p` results in the smallest expected number of dry commutes?

Chapter 7

Data Frames

Can one be a good data analyst without being a half-good programmer? The short answer to that is, 'No.' The long answer to that is, 'No.'

—Frank Harrell

Up to this point we have given a great deal of attention to vectors, and we have always treated them as one-dimensional objects: a vector has a length, but not a "width."

It is time to begin working in two dimensions. In this Chapter we will study *matrices*, which are simply vectors that have both length and width. Matrices are immensely useful for scientific computation in R, but for the most part we will treat them as a warm-up for *data frames*—the two-dimensional R-objects that are especially designed for the storage of data collected in the course of practical data analysis. Once you understand how to construct and manipulate data frames, you will be ready to learn how to visualize and analyze data using R.

7.1 Introduction to Matrices

In R, a *matrix* is actually an atomic vector—it can only hold one type of element—but with two extra attributes:

- a certain number of rows, and
- a certain number of columns.

One way to create is matrix is to take a vector and *give* it those two extra attributes, via the `matrix()` function. Here is an example:

```
numbers <- 1:24  # this is an ordinary atomic vector
numbersMat <- matrix(numbers, nrow = 6, ncol = 4)  # make a matrix
numbersMat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    2    8   14   20
## [3,]    3    9   15   21
## [4,]    4   10   16   22
## [5,]    5   11   17   23
## [6,]    6   12   18   24
```

Of course if you are making a matrix out of 24 numbers and you know that it's going to have 6 rows, then you know it must have 4 columns. Similarly, if you know the number of columns then the number of rows is determined. Hence you could have constructed the matrix with just one of the row or column arguments, like this:

```
numbersMat <- matrix(numbers, nrow = 6)
```

Notice that the numbers went down the first column, then down the second, and so on. If you would rather fill up the matrix row-by-row, then set the `byrow` parameter, which is `FALSE` by default, to `TRUE`:

```
matrix(numbers, nrow = 6, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
## [6,]   21   22   23   24
```

Sometimes we like to give names to our rows, or to our columns, or even to both:

```
rownames(numbersMat) <- letters[1:6]
colnames(numbersMat) <- LETTERS[1:4]
numbersMat
```

```
##   A  B  C  D
## a 1  7 13 19
## b 2  8 14 20
## c 3  9 15 21
## d 4 10 16 22
```

```
## e 5 11 17 23
## f 6 12 18 24
```

Matrices don't have to be numerical. They can be character or logical matrices as well:

```
creatures <- c("Dorothy", "Lion", "Scarecrow", "Oz",
               "Toto", "Boq")
matrix(creatures, ncol = 2)
```

```
##      [,1]        [,2]
## [1,] "Dorothy"   "Oz"
## [2,] "Lion"      "Toto"
## [3,] "Scarecrow" "Boq"
```

If you have to spread out the elements of a matrix into a one-dimensional vector, you can do so:

```
as.vector(numbersMat)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24
```

7.2 Matrix Indexing

Matrices are incredibly useful in data analysis, but the primary reason we are talking about them now is to get you used to working in two dimensions. Let's practice sub-setting with matrices.

We use the sub-setting operator `[` to pick out parts of a matrix. For example, in order to get the element in the second row and third column of `numbersMat`, ask for:

```
numbersMat[2,3]
```

```
## [1] 14
```

The row and column numbers are called *indices*.

If we want the entire second row, then we could ask for:

```
numbersMat[2,1:4]
```

```
##  A  B  C  D
##  2  8 14 20
```

The result is a one-dimensional vector consisting of the elements in the second row of `numbersMat`. It inherits as its names the column names of `numbersMat`.

Actually, if you want the entire row you don't have to specify which columns you want. Just leave the spot after the comma empty, like this:

```
numbersMat[2, ]
```

```
##  A  B  C  D
##  2  8 14 20
```

What if you want some items on the second row, but only the items in columns 1, 2 and 4? Then frame your request in terms of a vector of column-indices:

```
numbersMat[2, c(1, 2, 4)]
```

```
##  A  B  D
##  2  8 20
```

You can specify a vector of row-indices along with a vector of column-indices, if you like:

```
numbersMat[1:2, 1:3]
```

```
##   A B  C
## a 1 7 13
## b 2 8 14
```

If the vector has row or column names then you may use them in place of indices to make a selection:

```
numbersMat[, c("B", "D")]
```

```
##    B  D
## a  7 19
## b  8 20
## c  9 21
## d 10 22
## e 11 23
## f 12 24
```

You can use sub-setting to change the values of the elements of a matrix

```
numbersMat[2,3] <- 0
numbersMat
```

```
##   A  B  C  D
## a 1  7 13 19
## b 2  8  0 20
## c 3  9 15 21
## d 4 10 16 22
## e 5 11 17 23
## f 6 12 18 24
```

You can assign a value to an entire row:

```
numbersMat[2,] <- 0
numbersMat
```

```
##   A  B  C  D
## a 1  7 13 19
## b 0  0  0  0
## c 3  9 15 21
## d 4 10 16 22
## e 5 11 17 23
## f 6 12 18 24
```

In the code above, the 0 was "recycled" into each of the four elements of the second row

You can assign the elements of a vector to corresponding selected elements of a matrix:

```
numbersMat[2,] <- c(100, 200, 300, 400)
numbersMat
```

```
##     A   B   C   D
## a   1   7  13  19
## b 100 200 300 400
## c   3   9  15  21
## d   4  10  16  22
## e   5  11  17  23
## f   6  12  18  24
```

7.2.1 To Drop or Not?

Note that when we ask for a single row of `numbersMat` we got a regular one-dimensional vector:

```
numbersMat[3, ]
```

```
##  A  B  C  D
##  3  9 15 21
```

The same things happens if we ask for a single column:

```
numbersMat[ , 2]
```

```
##   a   b   c   d   e   f
##   7 200   9  10  11  12
```

We get the second column of `numbersMat`, but as a regular vector. It's not a "column" anymore. (Note that it inherits the row names from `numbersMat`.)

When a subset of a matrix comes from only one row or column, R takes the opportunity to "drop" the class of the subset from "matrix" to "vector." If you would like the subset to stay a vector, set the `drop` parameter, which by default is `TRUE`, to `FALSE`. Thus the second column of `numbersMat`, kept as a matrix with six rows and one column, is found as follows:

```
numbersMat[ , 2, drop = FALSE]
```

```
##     B
## a   7
## b 200
## c   9
## d  10
## e  11
## f  12
```

In most applications people want the simpler vector structure, so they usually leave `drop` at its default value.

7.3 Operations on Matrices

Matrices can be involved in arithmetical and logical operations.

7.3.1 Arithmetical Operations

The usual arithmetic operations apply to matrices, operating element-wise. For example, suppose that we have:

```
mat1 <- matrix(rep(1, 4), nrow = 2)
mat2 <- matrix(rep(2, 4), nrow = 2)
```

To get the sum of the above two matrices, R adds their corresponding elements and forms a new matrix out of their sums, thus:

```
mat1 + mat2
```

```
##      [,1] [,2]
## [1,]    3    3
## [2,]    3    3
```

R applies recycling as needed. For example, suppose we have:

```
mat <- matrix(1:4, nrow = 2)
mat
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

In order to multiply each element of `mat` by 2, we need not create a 2-by-2 matrix of 2's. We can simply multiply by 2, and R will take care of recycling the 2:

```
2 * mat
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

Or we could subtract 3 from each element of `mat`:

```
mat - 3
```

```
##      [,1] [,2]
## [1,]   -2    0
## [2,]   -1    1
```

7.3.2 Matrix Multiplication

This section is optional reading, but it may interest you if you know about matrix multiplication in linear algebra.

In order to accomplish matrix multiplication, we have to keep in mind that the regular multiplication operator * works element-wise on matrices, as we have already seen. For matrix multiplication R provides the special operator %*%. For example, consider the following matrices:

```
a <- matrix(1:6, ncol = 3)
a
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
b <- matrix(c(2, 1, -1), nrow = 3)
b
```

```
##      [,1]
## [1,]    2
## [2,]    1
## [3,]   -1
```

Observe that the number of columns of a is equal to the number of rows of b. Hence it is possible to form the matrix product a %*% b:

```
a %*% b
```

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

As expected, the result is a matrix having as many rows as the rows of aand as many columns as the columns of b.

It is also interesting to recall how matrix multiplication works when the second matrix has only one column. The product is obtained by multiplying each column of a by the element on the corresponding row of b, and adding the resulting matrices:

```
b[1,1]*a[ ,1, drop = FALSE] + b[2,1, drop = FALSE]*a[ ,2] + b[3,1]*a[ ,3, drop = FALSE]
```

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

7.3.3 Logical Operations

Boolean operations apply to matrices element-wise, just as they do to ordinary vectors. The result is a matrix of logical values. For examples, consider the original matrix numbersMat:

```
numbersMat <- matrix(1:24, nrow = 6)
```

Suppose we wish to determine which elements of numbersMat are odd. Then we simply ask whether the remainder of an element after division by 2 is equal to 1:

```
numbersMat %% 2 == 1
```

```
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE  TRUE  TRUE  TRUE
## [2,] FALSE FALSE FALSE FALSE
## [3,]  TRUE  TRUE  TRUE  TRUE
## [4,] FALSE FALSE FALSE FALSE
## [5,]  TRUE  TRUE  TRUE  TRUE
## [6,] FALSE FALSE FALSE FALSE
```

We can select elements from a matrix using a Boolean operator, too:

```
numbersMat[numbersMat %% 2 == 1]
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23
```

Note that the result is an ordinary, one-dimensional vector.

7.4 Introduction to Data Frames

R is known as a *domain-specific programming language*, meaning that although it can in principle perform any sort of computation that a human can perform (given enough pencil, paper and time), it was originally designed to perform tasks in a particular area of application. R's area of application is data analysis and statistics, especially when performed *interactively*—i.e., in a setting where the analyst asks for a relatively small computation, examines the results, modifies his or her requests and asks again, and so on.[1] Although R can be used effectively for a wide range of programming tasks, data analysis is where it really shines.

The data structures of R reflect its orientation to data analysis. We have met a data-oriented structure already—the table, which is one of many convenient ways to display the results of data analysis. For the purpose of organizing data in preparation for analysis, R provides the structure known as the *data frame*. A data frame facilitates the storage of related data in one location, in a form that makes the most sense to human users.

A data frame is like a matrix in that it is two-dimensional—it has rows and columns. Unlike a matrix, though, the elements of a data frame do not have to be all of the same data-type. Each column of a data frame is a vector—of the same length as all the others—but these vectors may be of different types: some numerical, some logical, etc.

7.4.1 Viewing a Data Frame

Let's take a close look at a data frame: the frame `m111survey`, which is available from the **tigerstats** package (Robinson and White, 2016). First let's attach the package itself:

```
library(tigerstats)
```

[1] Domain-specific languages (DSLs for short) stand in contrast to general-purpose programming languages that were designed to solve a wide variety of problems. Examples of important general-purpose languages include C and C++, Java, Python and Ruby. Although R is by now the one of the most widely-used DSLs in the world, there a number of other important ones, including Matlab, Octave and Julia for scientific computing, Emacs Lisp for the renowned Emacs editor, and SQL for querying databases. JavaScript is an interesting case: it started out as a DSL for web browsers, but has since expanded to power many web applications and is now being used to develop desktop applications as well.

In the R Studio IDE, we can get a look at the frame in a tab in the Editor pane if we use the `View()` function:

```
View(m111survey)
```

As with many objects provided by a package, we can get more information about it:

```
help("m111survey")
```

From the Help we see that `m111survey` records the results of a survey conducted in a number of sections of an elementary statistics course at Georgetown College. From the View we see that the frame is arranged in rows and columns. Each row corresponds to what in data analysis is known as a *case* or an *individual*: here, each row goes with a student who participated in the survey. The columns correspond to *variables*: measurements made on each individual. For a student on a given row, the values in the columns are the values recorded for that student.

When you are not working in R Studio, there are still a couple of way so view the frame. You could print it all out to the console:

```
m111survey
```

You could also use the `head()` function to view a specified number of initial rows:

```
head(m111survey, n = 6)  # see first six rows
```

7.4.2 The Stucture of a Data Frame

Further information about the frame may be obtained with the `str()` function:

```
str(m111survey)
```

```
## 'data.frame':    71 obs. of  12 variables:
##  $ height         : num  76 74 64 62 72 70.8 70 79 59 67 ...
##  $ ideal_ht       : num  78 76 NA 65 72 NA 72 76 61 67 ...
##  $ sleep          : num  9.5 7 9 7 8 10 4 6 7 7 ...
##  $ fastest        : int  119 110 85 100 95 100 85 160 90 90 ...
##  $ weight_feel    : Factor w/ 3 levels "1_underweight",..: 1 2 2 1 1 3 2 2 2 3 ...
##  $ love_first     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ extra_life     : Factor w/ 2 levels "no","yes": 2 2 1 1 2 1 2 2 2 1 ...
##  $ seat           : Factor w/ 3 levels "1_front","2_middle",..: 1 2 2 1 3 1 1 3 3 2 ...
##  $ GPA            : num  3.56 2.5 3.8 3.5 3.2 3.1 3.68 2.7 2.8 NA ...
##  $ enough_Sleep   : Factor w/ 2 levels "no","yes": 1 1 1 1 1 2 1 2 1 2 ...
##  $ sex            : Factor w/ 2 levels "female","male": 2 2 1 1 2 2 2 2 1 1 ...
##  $ diff.ideal.act.: num  2 2 NA 3 0 NA 2 -3 2 0 ...
```

The concept of *structure* extends far beyond the domain of computer programming.[2] In general the structure

[2] As an example outside of programming, consider what happens when you read a piece of literature "for structure." You begin by asking: "What kind of literature is this? Is it drama, a novel, or something else?" The answer lets you know what to expect as you read: if it's a novel, you know to suspend disbelief, whereas if it's a journalistic piece then you know to examine critically whatever it presents as fact. Next, you might outline the piece. When you make an outline, you are breaking the piece up into parts, and indicating how the parts relate to each other to advance the plot and/or message of the piece. Note that in the process of "reading for structure" you are following the pattern of the definition of structure offered above.

of any object consists of:

- the kind of thing that the object is;
- the parts of the object is made up of;
- the relationships between these parts—the rules, if you will, for how the parts work together to make the object do what it does.

In the case of `m111survey` the kind of thing this is its *class*: it's a data frame.

```
class(m111survey)
```

```
## [1] "data.frame"
```

Next we see the account of the parts of the object and the way in which the parts relate to one another:

```
## 71 obs. of  12 variables
```

From this we know that there are 71 individuals in the study. The data consists of 12 "parts"—the variables—which are related in the sense that they all provide information about the same set of 71 people.

After that the output of `str()` launches into an account of the structure of each of the parts, for example:

```
## $ height         : num  76 74 64 62 72 70.8 70 79 59 67 ...
```

We are told the kind of thing that height is: it's a numerical vector (a vector of type `double`, in fact). Next we are given the beginning of a statement of its parts: the heights of the individuals. So R is actually giving us the structure of the parts, as well as of the whole `m111survey`.

The variable `fastest` refers to the fastest speed—in miles per hour—that a person has ever driven a car. Note that it is a vector of type `integer`. Officially this is a numerical variable, too, but R is calling attention to the fact that the fastest-speed data is being stored as integers rather than as floating-point decimals.

The variables of a data frame are typically associated with the *names* of the frame:

```
names(m111survey)
```

```
##  [1] "height"          "ideal_ht"        "sleep"
##  [4] "fastest"         "weight_feel"     "love_first"
##  [7] "extra_life"      "seat"            "GPA"
## [10] "enough_Sleep"    "sex"             "diff.ideal.act."
```

By means of the names we can isolate a vector in any column, identified in our code in the format `frame$variable`. For example, to see the first ten elements of the `fastest` variable, we ask for:

```
m111survey$fastest[1:10]
```

```
##  [1] 119 110  85 100  95 100  85 160  90  90
```

In order to compute the mean fastest speed our subjects drove their cars, we can ask for:

```
mean(m111survey$fastest, na.rm = TRUE)
```

```
## [1] 105.9014
```

If you want to see the speeds that are at least 150 miles per hour, you could ask for:

```
m111survey$fastest[m111survey$fastest >= 150]
```

```
## [1] 160 190
```

If you worry that the form `frame$variable` will require an annoying amount of typing—as seems to be the case in the the example above—then you can use the `with()` function:

```
with(m111survey, fastest[fastest >=150])
```

```
## [1] 160 190
```

It's instructive to consider how `with()` works. If we were to includes the names of the parameters of `with()` explicitly, then the call would have looked like this:

```
with(data = m111survey, expr = fastest[fastest >=150])
```

For the `data` parameter we can supply a data frame or any other R-object that can be used to construct an environment . In this case `m111survey` provides a miniature environment consisting of the names of its variables. For the `expr` parameter we supply an expression for R to evaluate. As R evaluates the expression, it encounters names (such as `fastest`). Now ordinarily R would first search whatever counts as the active environment—in this case it's the Global Environment—for the names in the expression, but `with()` forces R to look first within the environment created by the `data` argument. In our example, R finds `fastest` inside `m111survey` and evaluates the expression on that basis. If it had not found `fastest` in `m111survey`, R would have moved on to the Global Environment and then the rest of the usual search path and (probably) would have found nothing, causing it to throw an "object not found" error message. In R, as in any other programming language, good programming depends very much on paying attention to how the language searches for the objects to which names refer.

7.4.3 Factors

Some of the variables in `m111survey` are called *factors*; an example is `seat`, which pertains to where one prefers to sit in a classroom:

```
str(m111survey$seat)
```

```
##  Factor w/ 3 levels "1_front","2_middle",..: 1 2 2 1 3 1 1 3 3 2 ...
```

Seating preference is an example of a *categorical* variable: one whose values are not meaningfully expressed in terms of numbers. When a categorical variable has a relatively small number of possible values, it can be convenient to store its values in a vector of class `factor`.

The *levels* of factor variable are its possible values. In the case of `seat`, these are: Front, Middle and Back. As a memory-saving measure, R stores the values in the factor as numbers, where 1 stands for the first level, 2 for the second level, and so on. But please bear in mind that we are dealing with a categorical variable, so the numbers don't relate to the possible values in any natural way: they are just storage conventions.

It's possible to create a factor from any type of vector, but most often this is done with a character vector. Suppose for instance, that eight people are asked for their favorite Wizard of Oz character and they answer:

```
ozFavs <- c("Glinda", "Toto", "Toto", "Dorothy", "Toto",
            "Glinda", "Scarecrow", "Dorothy")
```

We can create a factor variable as follows:

```
factorFavs <- factor(ozFavs)
factorFavs
```

```
## [1] Glinda    Toto      Toto      Dorothy   Toto      Glinda    Scarecrow
## [8] Dorothy
## Levels: Dorothy Glinda Scarecrow Toto
```

Note that the levels are given in alphabetical order: this is the default procedure when R creates a factor. It is possible to ask for a different order, though:

```
factor(ozFavs, levels = c("Toto", "Scarecrow", "Glinda", "Dorothy"))
```

```
## [1] Glinda    Toto      Toto      Dorothy   Toto      Glinda    Scarecrow
## [8] Dorothy
## Levels: Toto Scarecrow Glinda Dorothy
```

In many instances it is appropriate to convert a character vector to a factor, but sometimes this is not such a great idea. Consider something like your address, or your favorite inspirational quote: pretty much every person in a study will have a different address or favorite quote than others in the study. Hence there won't be any memory-storage benefit associated with creating a factor: the vector of levels—itself a character vector—would require as much storage space as the original character vector itself! In addition, we will see that the status of a variable as class "factor" can affect how R's statistical and graphical functions deal with it. It's not a good idea to treat a categorical variable as a factor unless its set of possible values is considered important.

We will think more about how to deal with factor variables later on, when we begin data analysis in earnest.

7.5 Creating Data Frames

There are many ways to create data frames in R. Here we will introduce just two ways.

7.5.1 Creation from Vectors

Whenever you have vectors of the same length, you can combine them into a data frame, using the `data.frame()` function:

```
n <- c("Dorothy", "Lion", "Scarecrow")
h <- c(58, 75, 69)
a <- c(12, 0.04, 18)
ozFolk <- data.frame(name = n, height = h, age = a)
ozFolk
```

```
##        name height   age
## 1   Dorothy     58 12.00
## 2      Lion     75  0.04
## 3 Scarecrow     69 18.00
```

Note that at the time of creation you can provide the variables with any names that you like. If later on you change your mind about the names, you can always revise them:

```
names(ozFolk)
```

```
## [1] "name"   "height" "age"
```

```
names(ozFolk)[2] <- "Height"  # "height" was at index 2"
ozFolk
```

```
##        name Height   age
## 1   Dorothy     58 12.00
## 2      Lion     75  0.04
## 3 Scarecrow     69 18.00
```

Let's check the structure of the frame we have made:

```
str(ozFolk)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Dorothy","Lion",..: 1 2 3
##  $ Height: num  58 75 69
##  $ age   : num  12 0.04 18
```

Maybe we would prefer that the `name` variable not be a factor. We have a couple of options to accomplish this.

1. We could coerce `names` to a character variable, and assign it to the data frame:

   ```
   ozFolk$name <- as.character(ozFolk$name)
   str(ozFolk)
   ```

   ```
   ## 'data.frame':    3 obs. of  3 variables:
   ##  $ name  : chr  "Dorothy" "Lion" "Scarecrow"
   ##  $ Height: num  58 75 69
   ##  $ age   : num  12 0.04 18
   ```

2. We could prevent `names` from being made into a factor at the time of creation:

   ```
   ozFolk <- data.frame(name = n, height = h, age = a,
                        stringsAsFactors = FALSE)
   str(ozFolk)
   ```

   ```
   ## 'data.frame':    3 obs. of  3 variables:
   ##  $ name  : chr  "Dorothy" "Lion" "Scarecrow"
   ##  $ height: num  58 75 69
   ##  $ age   : num  12 0.04 18
   ```

7.5.2 Creation From Other Frames

If two frames have the same number of rows, you may combine their columns to form a new frame with the `cbind()` function:

```
ozMore <- data.frame( color = c("blue", "red", "yellow"),
                      desire = c("Kansas", "courage", "brains"))
cbind(ozFolk, ozMore)
```

```
##        name height   age  color  desire
## 1   Dorothy     58 12.00   blue  Kansas
## 2      Lion     75  0.04    red courage
## 3 Scarecrow     69 18.00 yellow  brains
```

Similarly if two data frames have the same number *and type* of columns then we can use the `rbind()` function to combine them:

```
ozFolk2 <- data.frame(name = c("Toto", "Glinda"),
                      height = c(12, 66), age = c(3, 246),
                      stringsAsFactors = FALSE)
rbind(ozFolk, ozFolk2)
```

```
##        name height    age
## 1   Dorothy     58  12.00
## 2      Lion     75   0.04
## 3 Scarecrow     69  18.00
## 4      Toto     12   3.00
## 5    Glinda     66 246.00
```

Note: `cbind()` and `rbind()` work for matrices, too.

7.6 Subsetting Data Frames

Our study of sub-setting matrices can be applied to the selection of parts of a data frame. As with a vector, one or both of the dimensions of the frame can come into play.

We can create a new data frame consisting of any columns we like from the original frame:

```
df <- m111survey[, c("height", "ideal_ht")]
head(df)
```

```
##   height ideal_ht
## 1   76.0       78
## 2   74.0       76
## 3   64.0       NA
## 4   62.0       65
## 5   72.0       72
## 6   70.8       NA
```

If we select just one column, then the result is a vector rather than a data frame:

```
df <- m111survey[, "height"]
is.vector(df)
```

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

If for some reason you want to prevent this, set `drop` to `FALSE`:

```
df <- m111survey[, "height", drop =FALSE]
head(df)
```

```
##   height
## 1   76.0
## 2   74.0
## 3   64.0
## 4   62.0
## 5   72.0
## 6   70.8
```

You may select particular rows, too:

```
m111survey[10:15, c("height", "ideal_ht")]
```

```
##    height ideal_ht
## 10     67       67
## 11     65       69
## 12     62       62
## 13     59       62
## 14     78       75
## 15     69       72
```

You can even select some of the rows at random. Here is a random sample of size six:

```
n <- nrow(m111survey)
df <- m111survey[sample(1:n, size = 6, replace = FALSE), ]
df[c("sex", "seat")]  # show just two columns
```

```
##       sex     seat
## 47 female 2_middle
## 51 female 2_middle
## 34   male 2_middle
## 19 female  1_front
## 41 female 2_middle
## 61 female  1_front
```

Note the function `nrow()` that gives the number of rows of the frame. When we sample six items without replacement from the vector `1:n`, we are picking six numbers at random from the row-numbers of the vector. Specifying these six numbers in the selection operator `[` yields the desired random sample of rows.

7.6.1 Boolean Expressions

It is especially common to select rows by the values of a logical vector. For example, to select the rows where the fast speed ever driven is at least 150 miles per hour, try this:

```
df <- m111survey[m111survey$fastest >= 150, ]
df[, c("sex", "fastest")]  # show just two of the variables
```

```
##     sex fastest
## 8  male     160
## 32 male     190
```

When you are selecting rows it can be convenient to use the `subset()` function. The first argument to the function is the frame from which you plan to select, and the second is the Boolean expression by which to select:

```
df <- subset(m111survey, fastest >= 150)
df[, c("sex", "fastest")]
```

```
##     sex fastest
## 8  male     160
## 32 male     190
```

Note that we did not need to type `m111survey$fastest`: the first argument to `subset()` provides the environment in which to search for names that appear in the Boolean expression.

The Boolean sub-setting expressions can be quite complex:

```
df <- subset(m111survey, seat == "3_back" & height < 72 & sex == "female")
df[, c("sex", "height", "seat")]
```

```
##       sex height   seat
## 9  female     59 3_back
## 20 female     65 3_back
## 30 female     69 3_back
## 53 female     69 3_back
## 70 female     65 3_back
```

7.7 Ordering Data Frames

You can reorder as well as select. For example, the following code selects the first five rows of`m111survey` and then reverses them:

```
df <- m111survey[, c("height", "ideal_ht")]
dfRev <- df[5:1, ]
head(dfRev)
```

```
##   height ideal_ht
## 5     72       72
## 4     62       65
## 3     64       NA
## 2     74       76
## 1     76       78
```

If you want, you can even scramble the rows of the data frame in a random order:

```
n <- nrow(m111survey)
shuffle <- sample(1:n, size = n, replace = FALSE)
df <- m111survey[shuffle, ]
head(df[c("sex", "seat")])  #show just two columns
```

```
##       sex     seat
## 9  female   3_back
## 62 female  1_front
```

```
## 49   male 2_middle
## 27   male  1_front
## 66   male 2_middle
## 43 female  1_front
```

It is quite common to order the rows of a frame according to the values of a particular variable. For example, you might want to arrange the rows by `height`, so that the frame begins with the shortest subject and ends with the tallest.

Accomplishing this task requires a study of R's `order()` function. Consider the following vector:

```
vec <- c(15, 12, 23, 7)
```

Call `order()` with this vector as an argument:

```
order(vec)
```

```
## [1] 4 2 1 3
```

`order()` returns the indices of the elements of `vec`, in the following order:

- the index of the smallest element (7, at index 4 of `vec`);
- the index of the second-smallest element (12, at index 2 of `vec`);
- the index of the third-smallest element (15, at index 1 of `vec`);
- the index of the largest element (23, at index 3 of `vec`).

Can you guess the output of the following function-call without looking for the answer underneath?

```
vec[order(vec)]
```

```
## [1]  7 12 15 23
```

Sure enough, the result is `vec` sorted: from smallest to largest element.

Now the sorting of `vec` could have been accomplished with R's `sort()`function:

```
sort(vec)
```

```
## [1]  7 12 15 23
```

The power of `order()` comes with the rearrangement of rows of a data frame. In order to "sort" the frame from shortest to tallest subject, call:

```
df <- m111survey[order(m111survey$height), ]
head(df[, c("sex", "height")])  # to show that it worked
```

```
##       sex height
## 45 female     51
## 26 female     54
## 9  female     59
## 13 female     59
## 40 female     60
## 69 female     61
```

If you want to order the rows from tallest to shortest instead, then use the `decreasing` parameter, which by default is `FALSE`:

```
df <- m111survey[order(m111survey$height, decreasing = TRUE), ]
head(df[, c("sex", "height")])  # to show that it worked
```

```
##       sex height
## 8    male     79
## 14 female     78
## 1    male     76
## 58   male     76
## 34   male     75
## 54   male     75
```

Sometimes you want to order by two or more variables. For example suppose you want to arrange the frame so that the folks preferring to sit in front come first, followed by the people who prefer the middle and ending with the people who prefer the back. Within these groups you would like people to be arranged from shortest to tallest. Then call:

```
ordering <- with(m111survey, order(seat, height))
df <- m111survey[ordering, ]
head(df[, c("seat", "height")], n = 10)  # see if it worked
```

```
##       seat height
## 45 1_front     51
## 26 1_front     54
## 13 1_front     59
## 69 1_front     61
## 4  1_front     62
## 12 1_front     62
## 23 1_front     63
## 38 1_front     63
## 61 1_front     63
## 57 1_front     64
```

7.8 New Variables from Old

Quite often you will want to *transform* one or more variables in a data frame. Transforming a variable means changing its values in a systematic way.

For example, you might want to measure height in feet rather than inches. Then you want the following

```
heightInFeet <- with(m111survey, height/12)  # 12 inches in a foot
```

If you plan to use this new variable in your analysis later on, it might be a good idea to add it to the data frame:

```
m111survey$height_ft <- heightInFeet
```

Another common need is to *recode* the values of a categorical variable. For example, you might want to divide people into two groups: those who prefer to sit in the back and those who don't. This is a good time to use `ifelse()`:

```
seat2 <- ifelse(m111survey$seat == "3_back", "Back", "Other")
m111survey$seat2 <- seat2
```

If you plan to re-code into a variable that involves more than two values, then you might want to look into the `mapvalues()` function from the **plyr** package (Wickham, 2016):

```
seat3 <- plyr::mapvalues(m111survey$seat,
                from = c("1_front", "2_middle", "3_back"),
                to = c("Front", "Middle", "Back"))
str(seat3)
```

```
##  Factor w/ 3 levels "Front","Middle",..: 1 2 2 1 3 1 1 3 3 2 ...
```

The do-it-yourself approach is to write a loop. Remember `switch()`?

```
seat <- m111survey$seat
seat3 <- character(length(seat))  # this will be the recoded variable
for ( i in 1:length(seat) ) {
  seat3[i] <- switch(as.character(seat[i]),
                     "1_front" = "Front",
                     "2_middle" = "Middle",
                     "3_back" = "Back")
}
str(seat3)
```

```
##  chr [1:71] "Front" "Middle" "Middle" "Front" "Back" "Front" "Front" ...
```

The re-coding is done but the result is a character vector and not a factor. We have to make it a factor ourselves:

```
m111survey$seat3 <- factor(seat3, levels = c("Front", "Middle", "Back"))
```

This seems like a lot of work!

Another common transformation involves turning a numerical variable into a factor. For example, we might need to classify people as:

- Tall (height over 70 inches)
- Medium (65 - 70 inches)
- Short (less than 65 inches)

The `cut()` function will be helpful.

```
heightClass <- cut(m111survey$height,
                   breaks = c(-Inf, 65, 70, Inf),
                   labels = c("Short", "Medium","Tall"),
                   right = TRUE)
str(heightClass)
```

```
##  Factor w/ 3 levels "Short","Medium",..: 3 3 1 1 3 3 2 3 1 2 ...
```

Setting `right = TRUE` indicates that the upper bound of each interval is included in the interval. Thus, a person with a height of 70 inches is classed as Medium, not Tall.

7.8.1 Getting Rid of Variables

We have added several variables to `m111survey`. In order to remove them (or any other variables we don't want) we can assign them the value `NULL`.

```
names(m111survey)
```

```
##  [1] "height"          "ideal_ht"        "sleep"
##  [4] "fastest"         "weight_feel"     "love_first"
##  [7] "extra_life"      "seat"            "GPA"
## [10] "enough_Sleep"    "sex"             "diff.ideal.act."
## [13] "height_ft"       "seat2"           "seat3"
```

```
m111survey$height_ft <- NULL
m111survey$seat2 <- NULL
m111survey$seat3 <- NULL
names(m111survey)  # the extra variables are gone
```

```
##  [1] "height"          "ideal_ht"        "sleep"
##  [4] "fastest"         "weight_feel"     "love_first"
##  [7] "extra_life"      "seat"            "GPA"
## [10] "enough_Sleep"    "sex"             "diff.ideal.act."
```

Glossary

Matrix An atomic vector that has two additional attributes: a number of rows and a number of columns.

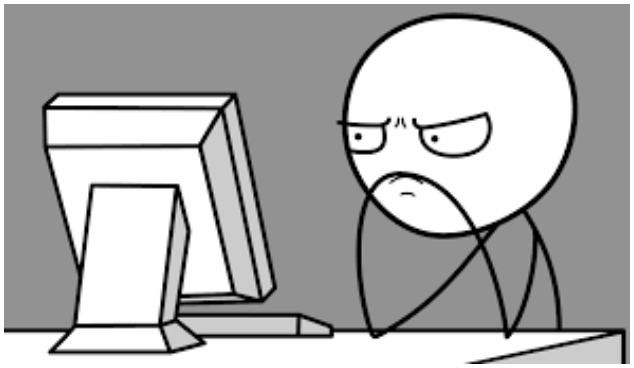
Data Frame A two-dimensional data structure in R in which the columns are atomic vectors that can be of different types.

Case (also called an Individual) An individual unit under study. In a data frame in R, the rows correspond to cases.

Variable (in Data Analysis) In data analysis, a *variable* is a measurement made on the individuals in a study.

Categorical Variable (in Data Analysis) In data analysis, a *categorical variable* is a variable whose values cannot be expressed meaningfully by numbers.

Exercises



1. R has a function called `t()` that computes the *transpose* of a given matrix. This means that it switches around the rows and columns of the matrix, like this:

```
myMatrix <- matrix(1:24, nrow = 6)
myMatrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    2    8   14   20
## [3,]    3    9   15   21
## [4,]    4   10   16   22
## [5,]    5   11   17   23
## [6,]    6   12   18   24
```

```
t(myMatrix)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    7    8    9   10   11   12
## [3,]   13   14   15   16   17   18
## [4,]   19   20   21   22   23   24
```

   Write your own function called `transpose()` that will perform the same task on any given matrix. The function should take a single parameter called `mat`, the matrix to be transposed. **Of course you may NOT use `t()` in the code for your function!**

   Hints: Your function will have to:

   1. break `mat` down into the vector of its elements, and then
   2. build the new matrix from those elements, with a number of rows equal to the number of columns of `mat`.

   For the first task, `as.vector()` will be useful. For the second task, recall that the function `nrow()` returns the number of rows of a given matrix. You will be interested to know that there is also a function `ncol()` that computes the number of columns of a given matrix.

2. R has functions called `rowSums()` and `colSums()` that will respectively sum the rows and the columns of a matrix. Here is an example:

```
myMatrix <- matrix(1:24, nrow = 6)
rowSums(myMatrix)
```

```
## [1] 40 44 48 52 56 60
```

Your task is to write your own function called `dimSum()` that will sum either the rows or the columns of a given matrix. The function should have two parameters:

- `mat`: the matrix to be summed.
- `dim`: the dimension to sum along, either rows or columns. The default value should be `c("rows", "columns")`, and you should use argument-matching (see Section 3.4.2) so that the user doesn't have to spell out all of the possible arguments.

You may NOT use `rowSums()` or `colSums()` in the code for your function. A typical example of use should look like this:

```
myMatrix <- matrix(1:24, nrow = 6)
dimSum(myMatrix, "c")
```

```
## [1]  21  57  93 129
```

Hint: To start, take it easy on yourself. Instead of writing a function that sums rows or columns as per the user's request, write a function called `myRowSums()` that will just find the row-sums of any given matric. You could use a loop, here. Your function can loop along the rows of the matrix, computing the row sums and storing them in a results vector that you set up prior to the loop.

Once you have `myRowSums()` working, write a function called `myColSums()` that find the column-sums of any given matrix.

Finally, use the two previously-created functions to write the function `dimsum()`.

3. Starting with `m111survey` in the **tigerstats** package, write the code necessary to create a new data frame called `smaller` that consists precisely of the male students who believe in extraterrestrial life and who are more than 68 inches tall. The new data frame should contain all of the original variables except for `sex` and `extra_life`.

4. Write a function called `dfRandSelect()` that randomly selects (without replacement) a specified number of rows from a given data frame. The function should have two parameters:

   - `df`: the data frame from which to select;
   - `n`: the number of rows to select.

   If `n` is greater than the number of rows in `df`, the function should return immediately with a message informing the user that the required task is not possible and informing him/her of the number of rows in `df`. Typical examples of use should be as follows:

```
dfRandSelect(tigerstats::fuel, 5)
```

```
##    speed efficiency
## 10   100       8.27
## 6     60       5.90
## 9     90       7.57
## 14   140      11.77
## 2     20      13.00
```

```
dfRandSelect(tigerstats::fuel, 200)
```

```
## No can do!  The frame has only 15 rows.
```

Hint: Use the function `nrow()`, which gives the number of rows of a matrix or data frame.

5. Create your own data frame, named `myFrame`. The frame should have 100 rows, along with the following variables:

   - `lowerLetters`: a character vector randomly-produced 3-letter strings, like "chj", "bbw", and so on. The letters should all be lowercase.
   - `height`: a numerical vector consisting of real numbers chosen randomly between the values of 60 and 75.
   - `sex`: a factor whose possible value are "female" and "male". Again, these values should be chosen randomly.

   A call to `str(myFrame)` would come out *like* this (although your results will vary a bit since the vectors are constructed randomly):

```
str(myFrame)
```

```
## 'data.frame':    100 obs. of  3 variables:
##  $ lowerLetters: chr  "hst" "stx" "knq" "bit" ...
##  $ height      : num  68.6 67 63.1 70.7 62 ...
##  $ sex         : Factor w/ 2 levels "female","male": 2 2 1 1 1 2 1 1 2 2 ...
```

   `summary()` is useful when working with data frames. Here is how a call to `summary(myFrame)` might look:

```
summary(myFrame)
```

```
##  lowerLetters          height          sex
##  Length:100         Min.   :60.63   female:46
##  Class :character   1st Qu.:64.14   male  :54
##  Mode  :character   Median :68.56
##                     Mean   :68.06
##                     3rd Qu.:71.91
##                     Max.   :74.58
```

   Hint: If you have a vector of three letters, such as

```
vec <- c("g", "a", "r")
```

   then you can paste them together as follows:

```
paste0(vec, collapse = "")
```

```
## [1] "gar"
```

6. Study the data frame `fuel` in the **tigerstats** package. Note that the fuel efficiency is reported as the number of liters of fuel required to travel 100 kilometers. Look up the conversion between gallons and liters and between kilometers and miles, and use this information to create a new variable called `mpg` that gives the fuel efficiency as miles per gallon. While you are at it, create a new variable `mph` that gives the speed in miles per hour. Finally, add these new variables to the `fuel` data frame.

7. (*) Use matrices to generalize the simulation in the Appeals Court Paradox (see Section 6.5). Your goal is to write a simulation function called `appealsSimPlus()` that comes with all the options provided in the text, but with additional parameters so that the user can choose:

   - the number of judges on the court;
   - the probability for each judge to make a correct decision;
   - the voting pattern (how many votes each judge gets).

   A typical call to the functions should look like this:

```
appealsSimPlus(reps = 10000, seed = 5252,
               probs = c(0.95, 0.90, 0.90, 0.90, 0.80),
               votes = c(2, 1, 1, 1, 0))
```

   In the above call the court consists of five judges. The best one decides cases correctly 95% of the time, three are right 90% of the time and one is right 80%of the time. The voting arrangement is that the best judge gets two votes, the next three get one vote each, and the worst gets no vote. Any voting scheme—even a scheme involving fractional votes—should be allowed so long as the votes add up to the number of judges.

   Here is a hint. When you write the function it may be helpful to use the fact that `rbinom()` can take a `prob` parameter that is a vector of any length. Here's an example:

```
results <- rbinom(6, size = 100, prob = c(0.10, 0.50, 0.90))
results
```

```
## [1] 20 49 94 15 50 88
```

   The first and fourth entries simulate a person tossing a fair coin 100 times when she has only a 10% chance of heads. The second and fifth entries simulate the same, when the chance of heads is 50%. The third and sixth simulate coin-tossing when there is a 90% chance of heads.

   If you would like to arrange the results more nicely—say in a matrix where each column gives the results for a different person—you can do so:

```
resultsMat <- matrix(results, ncol = 3, byrow = TRUE)
resultsMat
```

```
##      [,1] [,2] [,3]
## [1,]   20   49   94
## [2,]   15   50   88
```

   Of course judges don't flip a coin 100 times, they decide one case at a time. Suppose you have five judges with probabilities as follows:

```
probCorrect <- c(0.95, 0.90, 0.90, 0.90, 0.80)
```

   If you would like to simulate the judges deciding, say, 6 cases, try this:

```
results <- rbinom(5*6, size = 1, prob= rep(probCorrect, 6))
resultsMat <- matrix(results, nrow = 6, byrow = TRUE)
resultsMat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    0    1
```

```
## [2,]    0    1    1    1    1
## [3,]    1    1    1    1    1
## [4,]    1    1    1    1    1
## [5,]    1    1    1    1    1
## [6,]    1    1    1    1    0
```

When it comes to applying the voting pattern to compute the decision in each case, consider matrix multiplication. For example, suppose that the pattern is:

```
votes <- c(2, 1, 1, 1, 0)
```

Then make `votes` a one-column matrix and perform matrix multiplication:

```
correctVotes <- resultsMat %*% matrix(votes, nrow = 5)
correctVotes
```

```
##      [,1]
## [1,]    4
## [2,]    3
## [3,]    5
## [4,]    5
## [5,]    5
## [6,]    5
```

Think about how to encapsulate all of this into a nice, general simulation function.

Chapter 8

Graphics

*It's hard to succinctly describe how **ggplot2** works because it embodies a deep philosophy of visualisation.*

—Hadley Wickham (creator of **ggplot2**)

Now that we know a bit about how data can be stored and manipulated in data frames, we can begin to *analyze* data, so in this Chapter we will take a closer look at the aspect of data analysis known as *data visialization*. We'll become acquainted with the *grammar of graphics*, a general approach to visualization, and then learn how this approach is implemented in the **ggplot2** with which we have worked previously. Finally, we'll engage in a case study.

8.1 The Grammar of Graphics

A graph begins with data, and the data we work with will be tidy data that comes in a data frame. Leland Wilkinson's Grammar of Graphics (see (Wilkinson, 2005)) posits that most quantitative graphics constructed from a data frame can be understood in terms of a few basic elements. In our quite elementary introduction to the Grammar, the elements to which we will pay the most attention are as follows:

- *Glyphs*: the basic units of a graph. Glyphs represent cases in the data frame. Each glyph corresponds to one or more cases, but no two glyphs correspond to the same case.
- *Aesthetics*: perceptual properties of glyphs that are not the same for all glyphs but instead vary depending on the values of variables for the case (or cases) that each glyph represents.
- *Frame*: special aesthetics that relate the **position** of each glyphs in the graph to values of variables for the cases that the glyph represents.
- *Scales*: particular choices that determine the precise relationship between aesthetic properties and data values for glyphs.
- *Guides*: visual aids that help the human viewer to infer data values for cases from the aesthetic properties of the glyphs that represent them.

We will clarify these abstract ideas with a series of examples. Many of our examples will be drawn from the `m111survey` data frame in the **tigerstats** package.

```
library(tigerstats)
help(m111survey)
```

You will recall that the data frame records the results of a survey of 71 students at Georgetown College in Kentucky. Each case (row in the frame) corresponds to an individual student. See Table 8.1.

Table 8.1: The first few rows of the m111survey data frame. Five variables are shown.

sex	fastest	GPA	seat	weight_feel
male	119	3.56	1_front	1_underweight
male	110	2.50	2_middle	2_about_right
female	85	3.80	2_middle	2_about_right
female	100	3.50	1_front	1_underweight
male	95	3.20	3_back	1_underweight
male	100	3.10	1_front	3_overweight

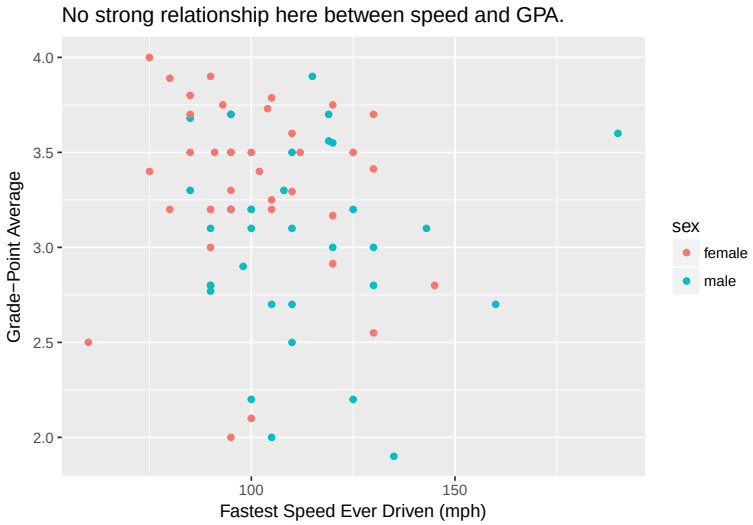


Figure 8.1: Scatterplot of fastest driving speed and GPA. Points are colored by sex of the student.

8.1.1 Example: a Scatterplot

We begin with a simple scatter plot based on the data. A scatter plot is often a good way to investigate graphically the relationship between two numerical variables. Figure 8.1 shows a scatter plot of student GPA vs. the fastest speed at which the student has ever driven a car.

8.1.1.1 The Glyphs

In this scatter plot, the glyphs are points. Each case—each student in the survey—is represented by one and only one point on the plot.

8.1.1.2 The Aesthetics

In ordinary discourse the term *aesthetic* refers to any perceptual property of an object. For a point, the list of its perceptual properties includes its location, its shape, its size, its color, and so on. In the Grammar of Graphics, however, only some of the properties—the one that vary from glyph to glyph depending on data—count as aesthetics in the graph.

For the scatter plot, the property of **size** is not considered to be an aesthetic: we can see that this is so because all of the points are the same size, and so the size cannot vary with the values of some variable in the data frame. The same goes for the property of **shape**: all of the points in this scatter plot are circular.

On the other hand, the property of **color** IS an aesthetic for the glyphs in the graph, since the males and the females in the study are represented by points of different colors. You could say that color is *mapped* to

the variable `sex` in the data frame:

- the reddish color goes with the value "female";
- the turquoise color goes with the value "male".

8.1.1.3 The Frame

In our scatter plot there are two other glyph properties that count as aesthetics:

- *x-location*: the position of the glyph relative to the horizontal axis of the graph;
- *y-location*: the position of the glyph relative to the vertical axis.

We can see that these properties are aesthetics because:

- *x-location* is mapped to the variable `fastest`: the further to the right the glyph is, the greater is the value of `fastest` for the student represented by that glyph.
- *y-location* is mapped to the variable `GPA`: the higher up the glyph is, the greater is the value of `GPA` for the student represented by that glyph.

Although x and y locations are just two more aesthetics, they are so crucial to the nature of a two-dimensional graph that they are classed separately in the Grammar of Graphics as the *frame* for the graph.

In the graphs we consider in this Chapter, the frame will always consist of at least the x-location, and sometimes—as in the case of our scatter plot—the y-location as well.

8.1.1.4 Scales

We can decide that color (for example) is to be mapped to `sex`, but that decision leaves open the question of how, precisely, to make the connection. The computer can make thousands of colors: which one will correspond to the value "male", and which to "female"? To answer that question is to choose a *scale*.

In this example our scale was:

- reddish = female
- turquoise = male

But we might have adopted a different scale, such as:

- blue = female
- pink = male

Every aesthetic mapping involves a choice of a scale. Consider x-location: apparently a point on the extreme left of the scatter plot represent a student who drove 50 miles per hour. A point at the extreme right corresponds to a speed of 200 miles per hour, and in general the relationship between x-location and `fastest` is linear: for example, a point halfway across the graph goes with a speed of 125 miles per hour, halfway between 50 mph and 200 mph. In the same way, the mapping of y-location to `GPA` involves a linear choice of scale.

8.1.1.5 Guides

How were we able to see what the scales were for each of the three aesthetic mappings in the scatter plot? We were assisted by three set of *guides*, one for each mapping:

- the legend to the right of the plot guided us from color to value of `sex`;
- the labels and tick marks on the x-axis and the thin vertical white lines guided us from x-location to value of `fastest`;
- the labels and tick marks on the y-axis and the thin horizontal white lines guided us from y-location to value of `GPA`.

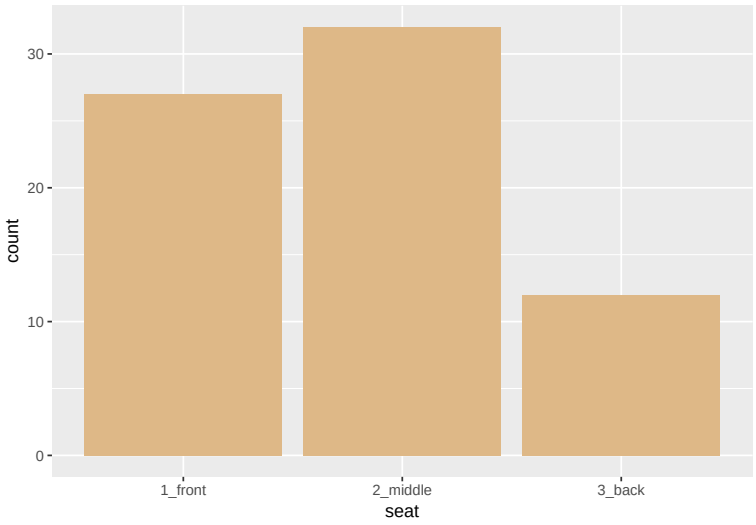


Figure 8.2: Bar graph of seating preference. The bars have a burlywood fill.

Most of the time, every aesthetic mapping is accompanied by a guide that gives the human viewer at least a rough idea of the scale chosen for that mapping.

8.1.1.6 Summary

In summary we say that for this scatter plot:

- The *glyphs* are points.
  - This time each glyph represents one and only one case.
- The *frame* is:
  - x = `fastest`
  - y = `GPA`
- Other *aesthetics* are:
  - color = `sex`
- There are *scales* for the three aesthetic mappings above. (But we usually don't say much about the x and y-location scales if they are linear, and we don't make a big deal of the color scale unless we went to some trouble to choose it ourselves.)
- The legend and the axis labels, tick marks and hash-lines are the *guides*.

8.1.2 Example: Two Bar Graphs

Bar graphs are useful in the study of categorical variables, especially factor variables that have only a few possible values. Figure 8.2 shows the distribution of the factor variable `seat` in the `mat111survey` data frame.

In this graph:

- The *glyphs* are bars.
  - This time each glyph an entire group of cases: there is a bar for the students who prefer the front, a bar for the students who prefer the middle, and a bar for the back-sitters.
- The *frame* is:
  - x = `seat`. Note that it is possible for x-location to map to a categorical variable!
  - In this graph the y-location does not count as part of the frame, since it is not really an aesthetic. Instead the height of a bar along the y-axis tells us how many students are represented by that

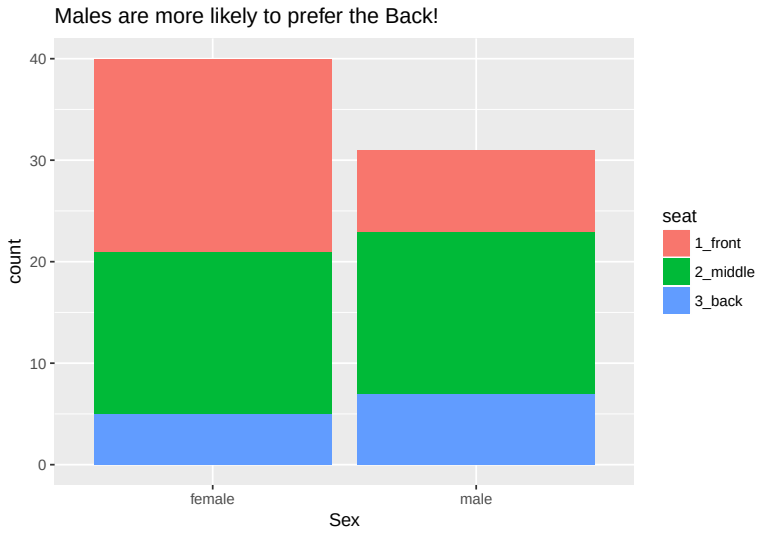


Figure 8.3: Seating preference, by sex.

bar. In the Grammar of Graphics we say that the y-axis represents a **statistic**—a value computed from the data. In the situation at hand, our statistic is a simple tally of the cases for each value of `seat`.

- There are no other *aesthetics*! The glyphs have various perceptual properties such as a shape rectangular and color, but these don't vary with the cases: the shape is always rectangular and the color is always burlywood.
- There is a *scale* for the x-location, but there is nothing very interesting about it: the three values of `seat` are equally spaced along the axis.
- There is a *guide* for the x-location: labels on the x-axis tell us which bar goes with which value of the variable `seat`.

Bar graphs can also be used to study the relationship between two categorical variables. Figure 8.3 shows the relationship between sex and seating preference in the `m111survey` data.

Again the glyphs are bars and each bar represents many cases, but now there is a bar for each combination of the values of `sex` and `seat`. The frame is again specified only by the x-location, but this time it is mapped to `sex`. There is another aesthetic as well: the color (more technically, the *fill*) of the bars is mapped to the variable `seat`, allowing us to see the relationship between the two categorical variables `sex` and `seat`. Scales and guides work much the same way as in the previous example.

8.1.3 Examples: Histograms, Density Plots and Box Plots

In this section we'll examine some glyphs that are useful in the visualization of numerical variables.

8.1.3.1 Histograms

How are the fastest speeds driven distributed, for students in the `m111survey` data? In order to investigate such a question graphically, we might make a *histogram* like the one in Figure 8.4.

In this graph:

- The *glyphs* are rectangles. Each rectangle represents cases where the value of `fastest` lies within a particular range covered by the bottom left and right corners of the rectangle.
- The *frame* is:
  - x = `fastest`.

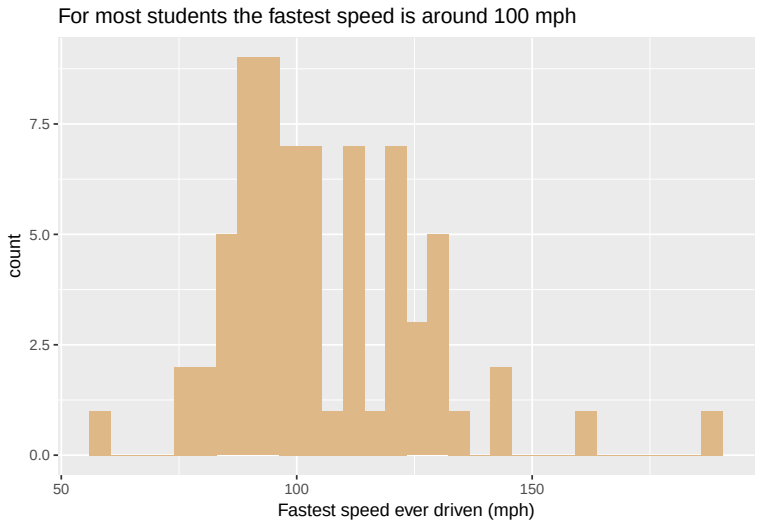


Figure 8.4: Histogram of the fastest speed ever driven.

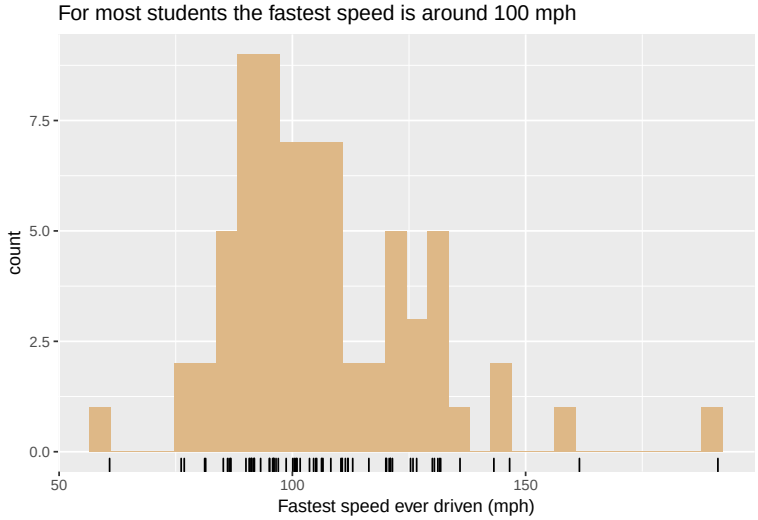


Figure 8.5: Histogram of the fastest speed ever driven.

  - As with our bar graphs, the y-location does not count as part of the frame, but instead represents a statistic, permitting the height of a rectangle to indicate the number of cases that it represents.
- Again there are no other *aesthetics*. The burlywood fill of the rectangles is constant.
- The *scale* for x-location, maps location to `fastest` in the familiar linear fashion, and the x-axis has the usual *guides* found for numerical variables.

Figure 8.5 is a variant, containing a second type of glyph: each student is now represented along the X-axis by a rug-tick located approximately at his or her fastest speed. (The ticks are actually "jittered" randomly so as to avoid over-plotting when two or more students report the same speed.) The addition of a second set of glyphs is called *layerng*, and is a common device to enhance the communicative power of a graph.

8.1.3.2 Density Plots

One may also study the distribution of numerical variable with a *density plot*, as in Figure 8.6. In this figure there is only one glyph, the curve itself, and it represents all of the cases. However, its height represents

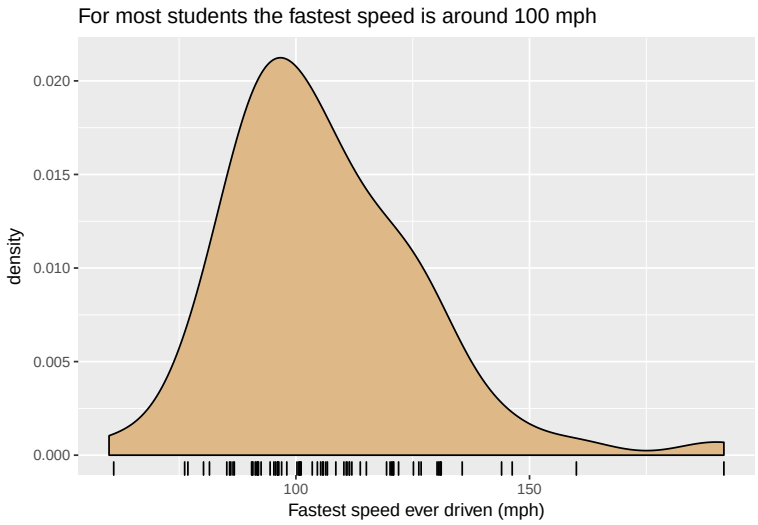


Figure 8.6: Density plot of the fastest speed ever driven.

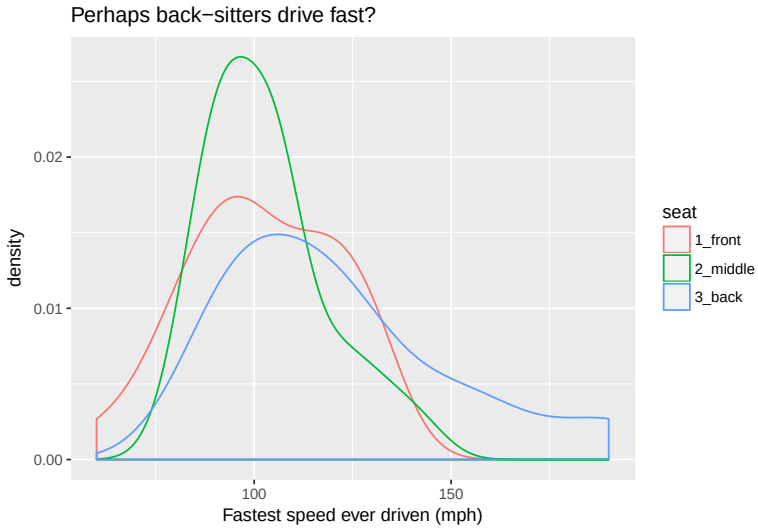


Figure 8.7: Density plot of the fastest speed ever driven.

crowding (density) of values of `fastest` for the cases: when the curve is high, many values are crowded closely together on the x-axis, and for speeds where the curve is low the viewer knows that few (if any) students drive around that speed. The y-axis is again used along with a statistic: for density plots the vertical scale is chosen in such a way that the total area under the density curve is 1, so that the area under the curve between two given speeds is approximately equal to the proportion of students who had speeds within that range. For density plots a rug, provided again by slightly jittered ticks, is a useful additional layer to indicate crowding of values.

Since they don't take up much territory on a graph, density curves are especially useful when we want to study the relationship between a numerical and a categorical variable. For example, Figure 8.7 shows density plots of the fastest speeds for each of the three possible seating preferences. The glyphs are again density curves, but since the color aesthetic has been mapped to `seat`, we get a separate and differently-colored glyphs, one for each seating-preference..

Another approach to the same graphing problem is to use a type of glyph known as a *violin*. Look at Figure 8.8.

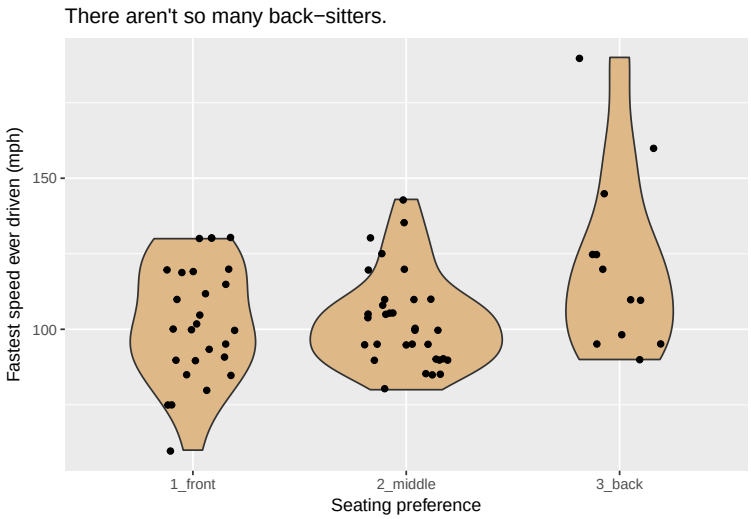


Figure 8.8: Violin plot of the fastest speed ever driven.

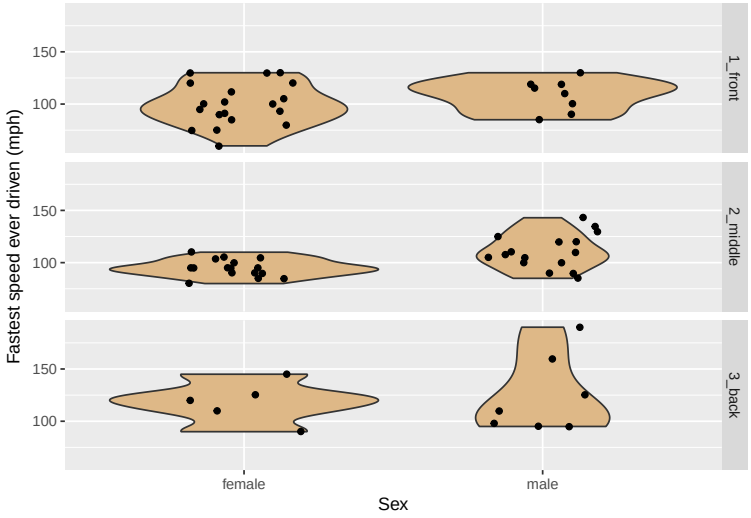


Figure 8.9: Violin plots of the fastest speed ever driven, by sex and seating preference.

A violin glyph is simply two mirror-images of the same density plot, pasted together along their bases. Thus the violin is thick where many values are clustered together and thin where data values are rare. In this plot, the frame is constituted by mapping x-location to `seat` and y-location to the variable `fastest`. For additional communicative power we have layered another set of glyphs–jittered points, one for each case–onto the plot.

Suppose that one wished to incorporate a third variable, such as sex, into the graph? One possible way to to this is to divide the graphs into separate plots based on the values of one of the categorical variables in question. The separate plots are known as *facets*, and are illustrated in Figure 8.9, where facet-ing has been done on the basis of the variable `seat`.

8.1.3.3 Box Plots

The *five number* summary is a convenient way to summarize the distribution of a numerical variable. The five numbers involved are:

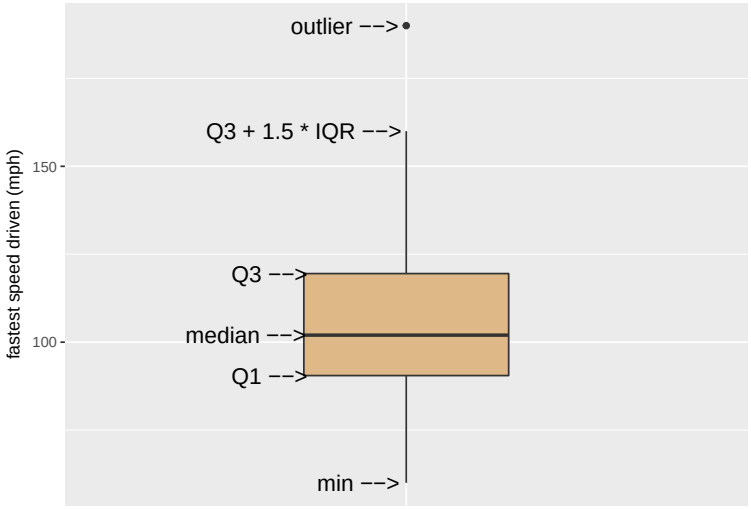


Figure 8.10: Illustration of a simple box plot.

- the minimum value
- the *first quartile* $Q1$, the 25^{th} percentile of the values
- the *median*, which is the 50^{th} percentile
- the *third quartile* $Q3$, the 75^{th} percentile
- the maximum value

Also of interest is the *interquartile range* IQR, defined as:

$$IQR = Q3 - Q1.$$

The interquartile range covers measures the spread in the middle 50% of the values.

In R the five number summary can be got quickly with the `fivenum()` function:

```
fnFastest <- fivenum(m111survey$fastest)
names(fnFastest) <- c("min", "Q1", "median", "Q3", "max")
fnFastest
```

```
##    min     Q1 median     Q3    max
##   60.0   90.5  102.0  119.5  190.0
```

A box-plot glyph is the graphical counterpart of the five number summary. Figure 8.10 shows how it works for the variable `fastest` in the `m111survey` data frame. The box ranges from $Q1$ to $Q3$, covering the middle half of the speeds. The lower hinge extends from $Q1$ down to the minimum speed. The upper hinge would have extended from $Q3$ to the maximum value, but the maximum value was flagged as an outlier. When **ggplot2** makes a box-plot, any point that is

- greater than $Q3 + 1.5 \times IQR$ or
- less than $Q1 - 1.5 \times IQR$

is flagged for individual plotting, and the corresponding hinge will be $1.5 \times IQR$ units long.

A single box glyph on its own is not very interesting. Where box plots shine is in the study of the relationship between a numerical variable and a categorical with a large number of levels, as in Figure **??**fig:utitlitybills1). Here the glyphs are boxes, with each box being constructed from the bills that were issued in a particular month.

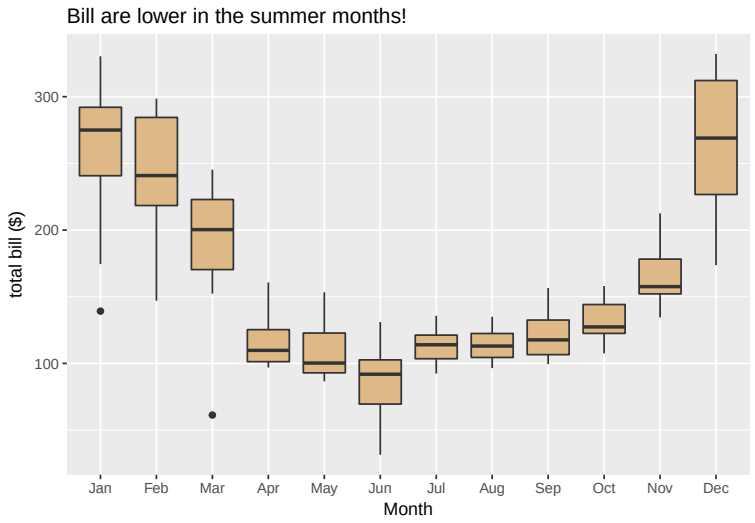


Figure 8.11: Utility bills throught the year.

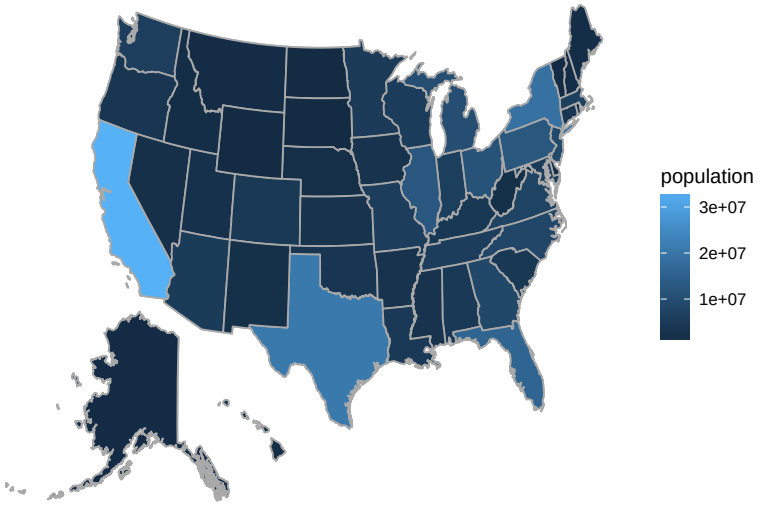


Figure 8.12: Choropleth map of state populations in the U.S.

8.1.4 Example: Choropleth Maps

The term "choropleth" derives from Greek and means "many regions." A choropleth graph is a graph in which the frame is provided by some sort of map with regions that might be countries, cities or counties in the U.S. etc.

In the choropleth map shown in Figure 8.12, is based on a data frame in which ease case is a state in the U.S (along with the District of Columbia). One of the variables is `population`, the population of the state. The glyphs are the territories of each of the U.S. states. The frame is determined by mapping x and y-location to latitude and longitude. The aesthetic property *color* is mapped to the `population`, and a guide is provided to the right of the graph.

Figure 8.13: A completely blank plot!

8.2 Implementation in ggplot2

In its syntax, the **ggplot2** package attempts to follow the Grammar of Graphics fairly closely. Let's see how this works by building up, in step-by-step fashion, to our initial graph example—the scatter plot in Figure 8.1.

8.2.1 Basic Setup: the Data Frame

Construction of a graph in **ggplot2** begins with the `ggplot()` function. The first parameter of the function is `data`, the value of which will be the data frame on which we plan to build the graph.

It is possible call `ggplot()` with just the data, and indeed it is instructive to do so. The result is seen in Figure 8.13: it is simply a blank window.

```
ggplot(data = m111survey)
```

8.2.2 More Setup: Establishing the Frame

`ggplot()` has a second parameter, the parameter `mapping`. Usually it is assigned the result of a call to the **ggplot** function `aes()`, which is used to establish aesthetic mappings. The common procedure is to use this first call to `aes()` to establish the frame: later calls will map other aesthetics properties to other variables, as desired.

If we want to work toward Figure 8.1, we will have to map x-location to `fastest` and y-location to `GPA`. This is accomplished by the following code:

```
ggplot(data = m111survey,
       mapping = aes(x = fastest, y = GPA))
```

The result appears as Figure 8.14. A frame has been established, along with guides to **ggplot2**'s default choice of linear scales for the mappings to `fastest` and to `GPA`.

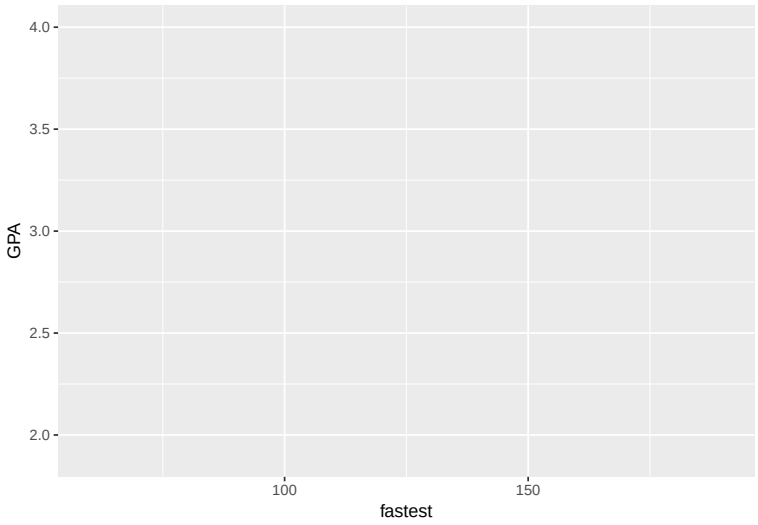


Figure 8.14: Just the frame: no glyphs yet!

It is worth noting that most programmers do not bother to name the `data` and `mapping` parameters. Figure 8.14 could just as well have been produced as follows:

```
ggplot(m111survey, aes(x = fastest, y = GPA))
```

In the future we will also drop these parameter names.

8.2.3 Labels

At any point we can add labels to our plot. If you are simply exploring data you don't need labels, but if you are writing up the final version of a report you will want to give careful consideration to labeling axes and to providing a good title (or—if you are able to to do so—a good caption). The following code adds labels for the x and y axes, a title—and even a subtitle, although subtitles are somewhat rare in practice.

```
ggplot(m111survey, aes(x = fastest, y = GPA)) +
  labs(x = "fastest speed ever driven (mph)",
       y = "grade point average",
       title = "Dunno what to say, yet!",
       subtitle = paste0('A title should briefly indicate the main',
                         ' lesson the plot has for us.  Instead of:',
                         ' "Scatterplot of\nGPA vs. fastest speed",',
                         ' say something like: "Speed and GPA are',
                         ' not strongly related."'))
```

8.2.4 Adding a Type of Glyph

It is high time now to make some data appear on our plot, so let's add some glyphs. In **ggplot2** syntax glyphs are added with function whose names are of the form:

```
geom_gylphType()
```

Thus we have such things as:

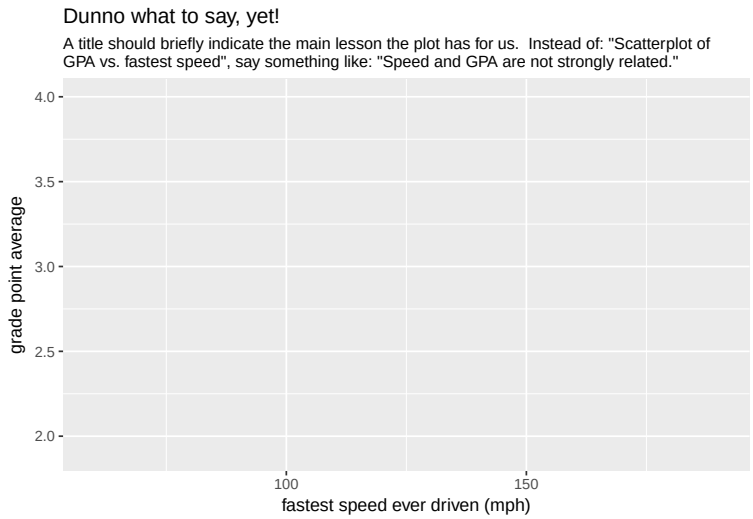


Figure 8.15: You can always add some labels!

- `geom_point()` for points;
- `geom_bar()` for the bars of a bar graph;
- `geom_histogram()` for the rectangles that make up a histogram;
- `geom_density()` for the curve of a density plot;
- 'geom_violin() for the violins of a violin plot;
- `geom_jitter()` for jittered points representing individual cases;
- `geom_rug()` for rug-ticks representing individual cases;
- and a number of other `geom`'s!

8.2.4.1 Our First Geoms

Let's add some points to the plot with the following code. The result appears as Figure 8.16. We are now quite close to the target plot.

```
ggplot(m111survey, aes(x = fastest, y = GPA)) +
  geom_point() +
  labs(x = "fastest speed ever driven (mph)",
       y = "grade point average",
       title = "Speed and GPA are not strongly related.")
```

8.2.4.2 Further Aesthetic Mappings

The final step in our first example is to map the *color* property of points to the variable `sex`. We do so by a call to `aes()`. Conventionally a mapping for a glyph is accomplished inside the `geom`-function that creates the glyph, as in the code below that creates figure 8.17, our target plot:

```
ggplot(m111survey, aes(x = fastest, y = GPA)) +
  geom_point(aes(color = sex)) +
  labs(x = "fastest speed ever driven (mph)",
       y = "grade point average",
```

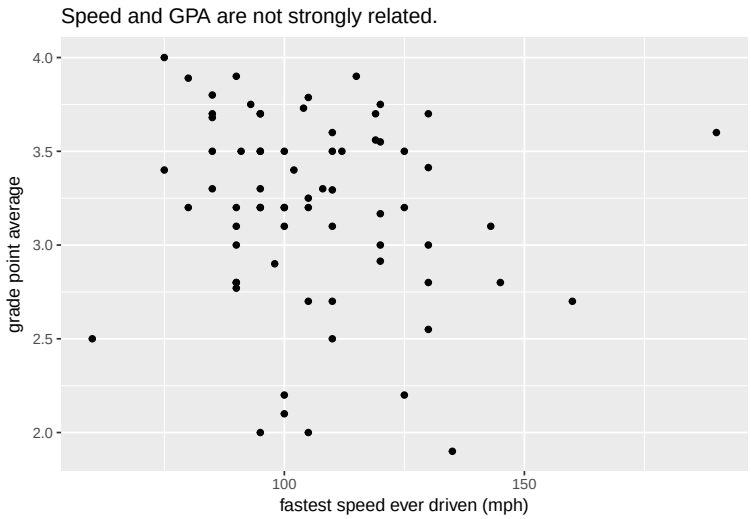


Figure 8.16: Finally: the data appears!

Figure 8.17: This is the target plot!

```
       title = "Speed and GPA are not strongly related.",
       subtitle = "(But guys tend to drive faster, and to have lower GPAs.)")
```

Points have other perceptual properties in besides their color. *Shape* is such a property. (From the point of view of the Grammar of Graphics, a point in itself is only an abstract location in space. Only when it assumes all of its perceptual properties does it actually "appear", and when it does appear it shape may be other than circular, just as its color may be other than, for instance, black.) Thus alternative way to incorporate sex into the graph would have been to map shape to `sex`,as in the following code that results in Figure 8.18:

```
ggplot(m111survey, aes(x = fastest, y = GPA)) +
  geom_point(aes(shape = sex)) +
  labs(x = "fastest speed ever driven (mph)",
```

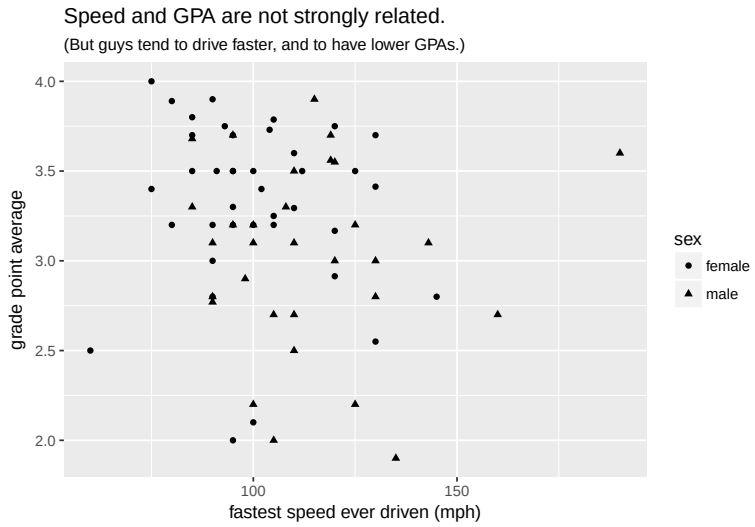


Figure 8.18: Mapping shape to the variable sex.

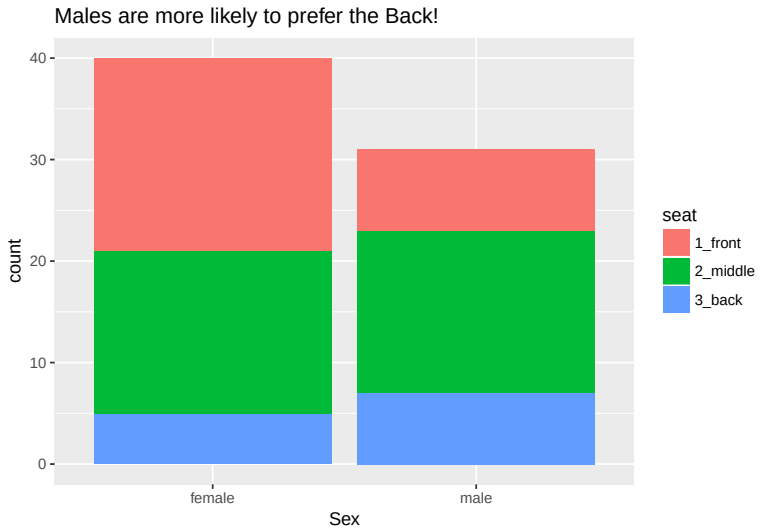


Figure 8.19: Seating preference, by sex.

```
       y = "grade point average",
       title = "Speed and GPA are not strongly related.",
       subtitle = "(But guys tend to drive faster, and to have lower GPAs.)")
```

Passing now from our scatter plot example, we consider how to achieve another of the plots studied in the previous section, namely Figure 8.3. Following the same logic of calls to `ggplot()` and a `geom`-function, we see that the bar graph on sex and seating preference can be obtained by mapping the *fill* property of bars to `seat` as seen in the following code (results shown again as Figure ‘8.19)

```
ggplot(m111survey, aes(x = sex)) +
  geom_bar(aes(fill = seat)) +
  labs(x = "Sex",
       title = "Males are more likely to prefer the Back!")
```

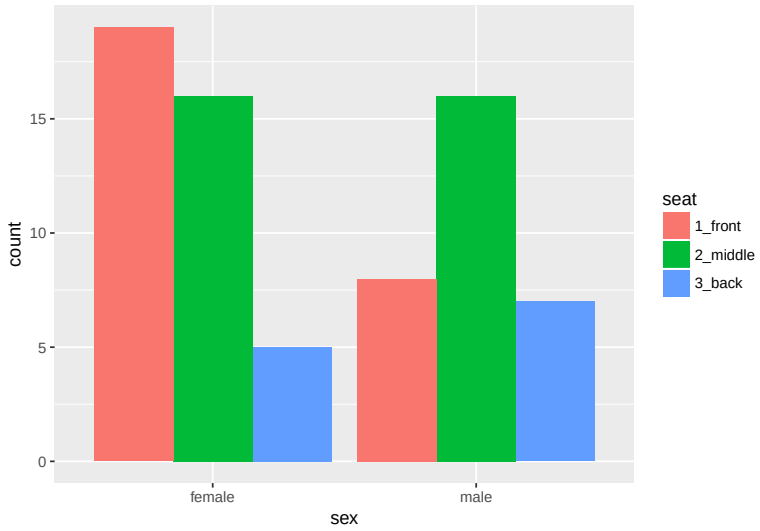


Figure 8.20: Seating preference, by sex—no stacking of bars..

Some people don't like the glyphs "stacked" in bar graphs. In order to mollify them we can set `position` to "dodge", as in the code below. The results appear in Figure 8.20.

```
ggplot(m111survey, aes(x = sex)) +
  geom_bar(aes(fill = seat), position ="dodge")
```

Note that *position* is not an aesthetic property: all of the bar dodge each other. Dodginess is not something that varies from glyph to glyph according to values in the data.

8.2.4.3 Aesthetic Mappings vs. Fixed Properties

It is wise to dwell a bit on the distinction between aesthetic mappings on the one and hand and fixed properties of glyphs on the other hand.

The key is this:

- An aesthetic mapping are always accomplished as an argument in a call to `aes()`. In this argument, the aesthetic property is the parameter name, and a variable is assigned to it, thus:
  - `geom_bar(aes(fill = seat))`
  - `geom_point(aes(color = sex))`
- A fixed property is determined by an argument to a `geom`-function call. The property to be fixed is the name of the parameter, and its constant value is the value supplied, thus:

Code	Effect
`geom_point(color = "blue")`	all the points are blue
`geom_point(shape = 22)`	all points are solid squares
`geom_point(size = 3)`	all points are bigger than default size(1)
`geom_bar(fill = "burlywood")`	all bars have the burlywood fill-color

As an example, let's code up the density plot of fastest speeds that occurred in Figure 8.4. The code is shown below and appears as 8.21

```
ggplot(m111survey, aes(x = fastest)) +
  geom_histogram(fill = "burlywood") +
  labs(x = "Fastest speed ever driven (mph)",
            title = "For most students the fastest speed is around 100 mph")
```

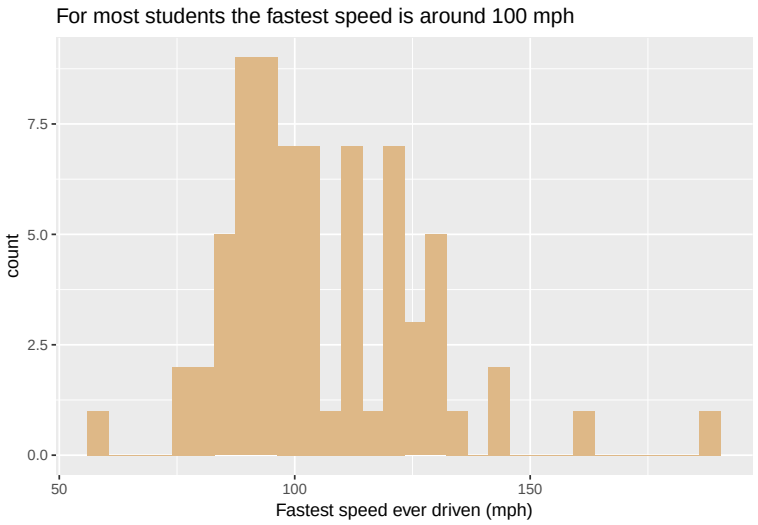


Figure 8.21: Histogram of the fastest speed ever driven. The fill-property of the curve is fixed to the ever-popular 'burlywood' color.

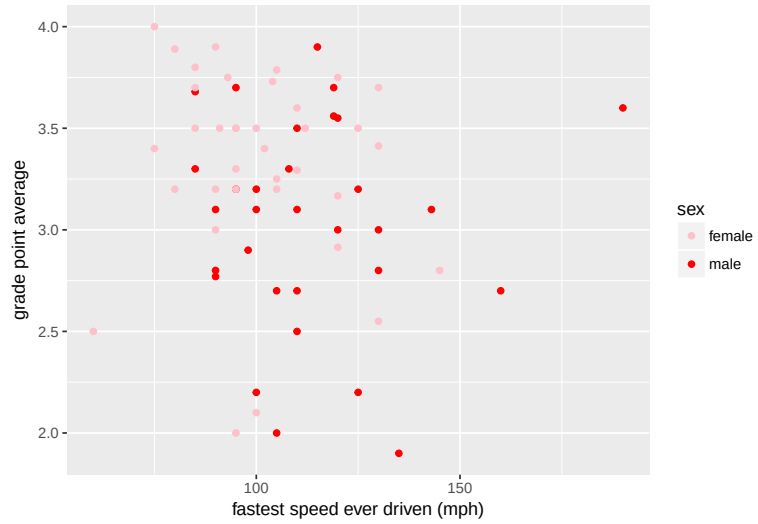


Figure 8.22: Color scale adjusted manually.

- and many others!

Here is one simple example of setting our own scales. The code below produces Figure 8.22, where we have set that "female" should go with pink and "male" with red.

```
ggplot(m111survey, aes(x = fastest, y = GPA)) +
  geom_point(aes(color = sex)) +
  scale_color_manual(values = c("pink", "red")) +
  labs(x = "fastest speed ever driven (mph)",
       y = "grade point average")
```

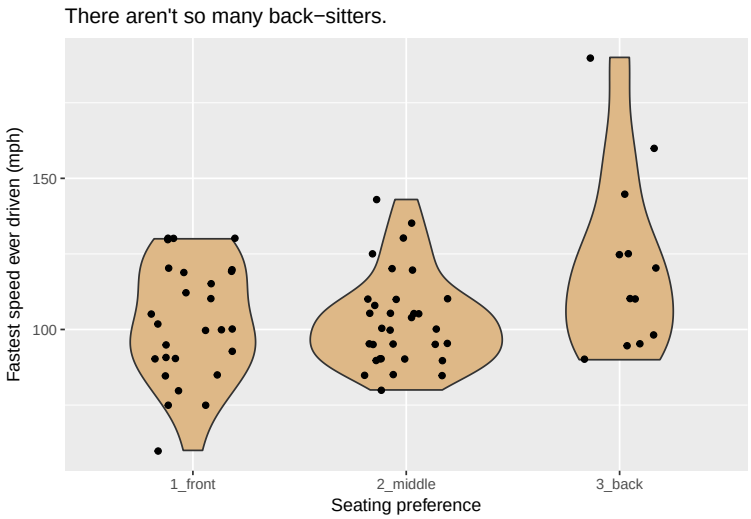


Figure 8.23: Violin plot of the fastest speed ever driven.

8.2.5 Layering: Adding Another Glyph Type

If you want to add another layer of glyphs, simply add on another call to a `geom`-function. In order to produce Figure 8.23, for example, we use the code below:

Note that the `width` parameter in the call to `geom_jitter()` determines how much the points are allowed to jitter horizontally.

8.2.5.1 Jitter-It-Yourself (JIY)

"Rug" glyphs are excellent, in especially in conjunction with density curves, but they have a downside. Consider, for example Figure 8.24 produced by the code below. When you examine the plot you will see that there aren't as many rug-ticks as there are students in the 1m111survey1 data. Many students reported driving at the same speed, so their rug-ticks plotted over each other.

```
ggplot(m111survey, aes(x = fastest)) +
  geom_density(fill = "burlywood") +
  geom_rug() +
  labs(x = "Fastest speed ever driven (mph)",
            title = "For most students the fastest speed is around 100 mph")
```

It would be nice to solve the problem by jittering the rug-ticks, but unfortunately rug-ticks don't jitter nicely on their own. One reasonable workaround is to create one's own randomly-jittered speeds and map the x-location of the rug-ticks to the new variable that holds the jittered values. The code below shows implements this idea, and results in Figure 8.25.

```
n <- nrow(m111survey)
m111survey$jitteredSpeeds <- m111survey$fastest + runif(n, 0, 2)
ggplot(m111survey, aes(x = fastest)) +
  geom_density(fill = "burlywood") +
  geom_rug(aes(x = jitteredSpeeds)) +
  labs(x = "Fastest speed ever driven (mph)",
            title = "For most students the fastest speed is around 100 mph")
```

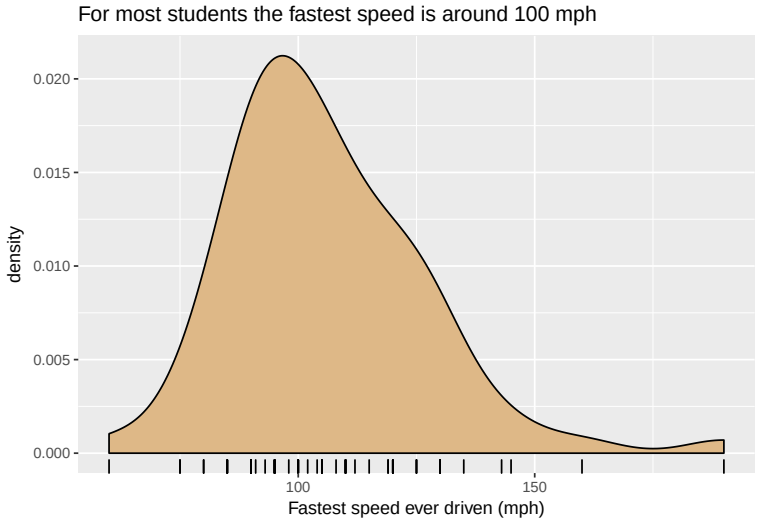


Figure 8.24: Density plot of the fastest speed ever driven. Some rug glyphs overplot each other.

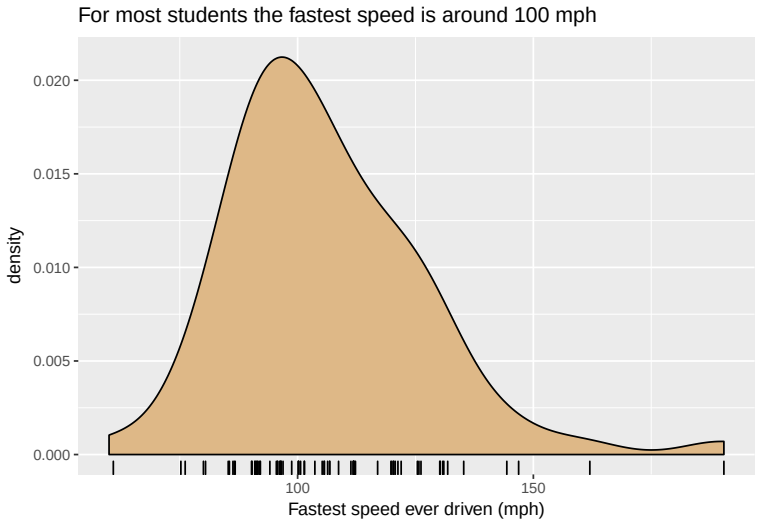


Figure 8.25: Density plot of the fastest speed ever driven. Rug glyphs are jittered.

8.2.6 Facets

As you will recall, a graph has facets when it is sub-divided into plots with one plot for each of the values of a categorical variable. **ggplot2** has two functions to manage facet-ing:

- `facet_grid()`
- `facet_wrap()`

We'll examine both of these functions in the context of some data frames from the **mosaicData** package, so you want to make sure to attach it to your search path:

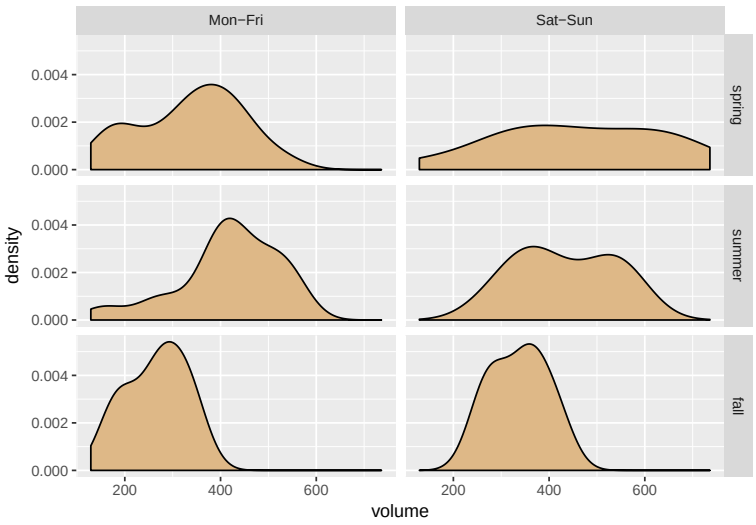


Figure 8.26: Volume of daily trail usage, by seaon and time of week.

```
library(mosaicData)
```

8.2.6.1 `facet_grid()`

The data frame `mosaicData::RailTrail` has information on usage of a converted railroad trail every day from April 5 to November 15, 2005. Study the Help file:

```
help(RailTrail)
```

Every row in the data frame represents a particular day between April 5 and November 15. Our goal is to study how the season (Spring, summer or Fall) and the time of week (weekday Mon-Fri vs. weekend Sat-Sun) relate to `volume` the number of people who use the trail on a given day.

Infromation about the season in which the day occurs is spread over the three 0-1 numerical variables `spring`, `summer` and `fall`. We propose to construct a single factor variable `season` variable from these three variables, as follows:

We'll also make a variant of the "0"-"1" character-based variable `weekday` that has more informative values:

```
RailTrail$wkday <- ifelse(RailTrail$weekday == "1",
                          "Mon-Fri", "Sat-Sun")
```

We are now ready make our graph. One possibility is to make a separate density plot of `volume` for each of the six possible combinations of values of the `season` and `wkday` variables. With `facet_grid()` we can arrange the plots in a grid so that the value of `season` is constant along rows and the value of `wkday` is constant along columns. This is accomplished by the following code, and the result appears as Figure 8.26.

```
ggplot(RailTrail, aes(x = volume)) +
  geom_density(fill = "burlywood") +
  facet_grid(season ~ wkday)
```

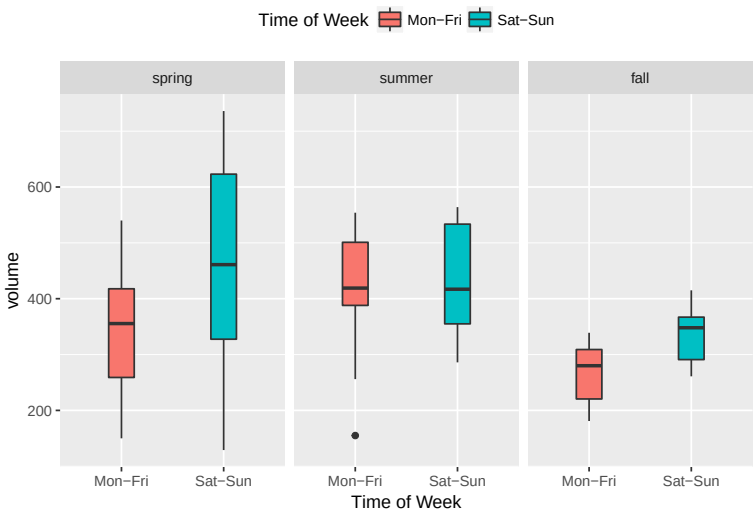


Figure 8.27: Striking a good balance between facet-ing and aesthetic mapping.

Of course our aim is to see how volume varies with season and time of week, but the horizontal orientation of the `volume` variable in the above graph makes comparison difficult for most human viewers. Perhaps facet-ing in two dimensions was a bit too much, in this situation. In the code below, we produce a one-row, three-column layout in which each facet corresponds to one of the three seasons.[1] Within each facet, the days are broken down by time of week and volumes are compared with boxplots. The result is seen in Figure 8.27. In the effort to incorporate the factor variables `season` and `wkday` into the graph, this second approach appears to strike a good balance between facet-ing and aesthetic mapping.

```
ggplot(RailTrail, aes(x = wkday, y = volume)) +
  geom_boxplot(aes(fill = wkday)) +
  facet_grid( . ~ season) +
  labs(x = "Time of Week",
       fill = "Time of Week") +
  theme(legend.position = "top", legend.direction = "horizontal")
```

8.2.6.2 `facet_wrap()`

Frequently it happens that one desires to facet by a single categorical variable, and the number of levels of a factor variable is too large for the entire graph to be displayed well along a single row or a single column. In that event, use `facet_wrap()`.

Consider, for example, the data frame `mosaicData::CPS85`, and suppose that we want to compare the ages of workers in the eight different sectors of employment. Eight is a rather large number of plots, so we facet in "wrap-style" by means of the code below. The resulting plot appears as Figure 8.28.

```
ggplot(CPS85, aes(x = age)) +
  geom_density(fill = "burlywood") +
  facet_wrap(~ sector, nrow = 3)
```

Once again, though, it may be wise to consider an approach that involves aesthetic mapping. When the number of levels is large, violin plots or boxplots may better approaches for representing a numerical variable such as `age`, as in the code below. The reuslting graph is shown in Figure 8.29.

[1] Note that the formula `season ~ .` in the call to `facet_grid()` would have produced a three-row, one-column layout.

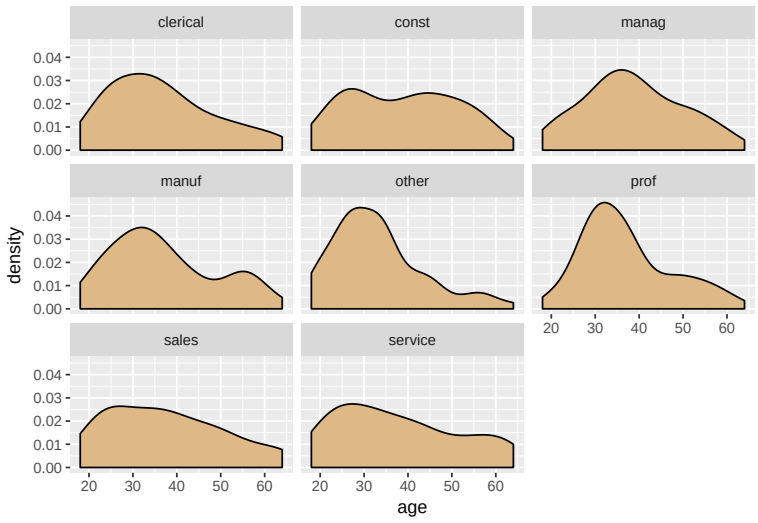


Figure 8.28: Age by sector.

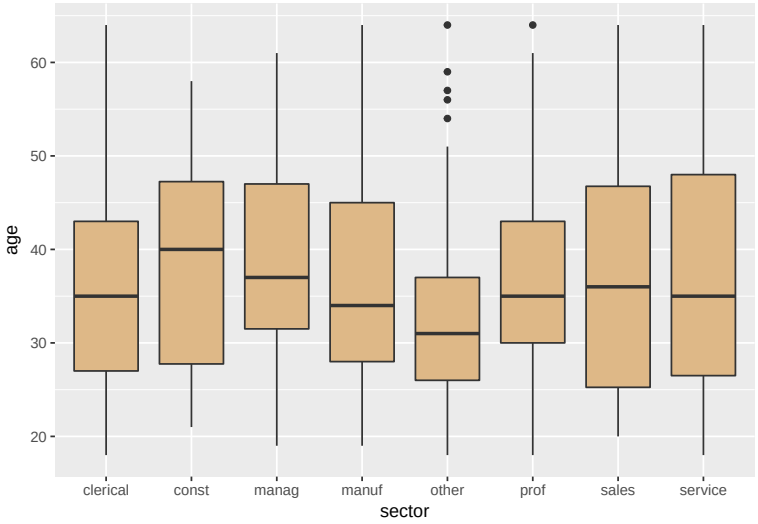


Figure 8.29: Aesthetic mapping is probably superior to facetting in this case.

```
ggplot(CPS85, aes(x = sector, y = age)) +
  geom_boxplot(fill = "burlywood")
```

In the exercises of this Chapter we will meet a case in which "wrap-style" facet-ing is quite useful.

8.3 A Case Study: US Births

In Section 1.2.5, we made a plot of the number of births in the United States for each day of that year (see Figure 8.30). We noticed that there appear to be two clouds of points. What accounts for this phenomenon? By now we have the R-programming chops to take on this question.

To begin with, look at all of the variables available in the data frame `Births78`:

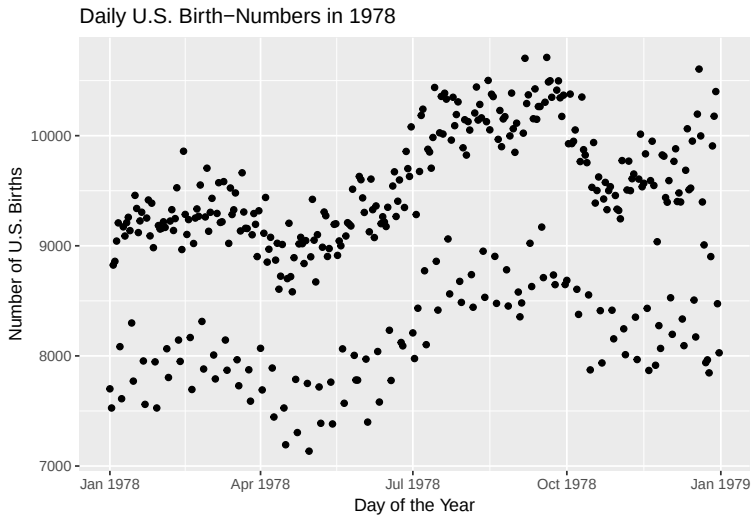


Figure 8.30: Some of the days have significantly fewer births. What's going on?

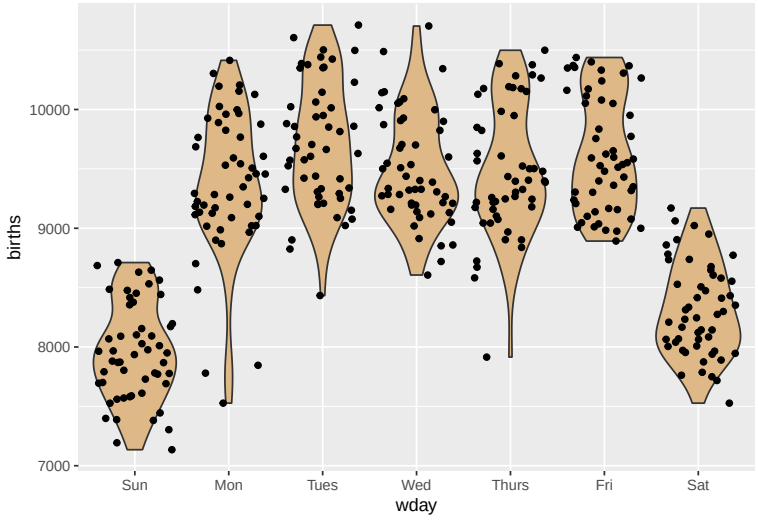


Figure 8.31: Violin plot of births, by day of the week.

```
str(Births78)
```

```
## 'data.frame':    365 obs. of  4 variables:
##  $ date     : POSIXct, format: "1978-01-01" "1978-01-02" ...
##  $ births   : int  7701 7527 8825 8859 9043 9208 8084 7611 9172 9089 ...
##  $ dayofyear: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ wday     : Ord.factor w/ 7 levels "Sun"<"Mon"<"Tues"<..: 1 2 3 4 5 6 7 1 2 3 ...
```

We see that the variable `wday` gives the name of the day of the week, for each of the days in the year. On a hunch, we make violin plots of the births for each of the days of the week. The code appears below, and the resulting plot is shown in Figure 8.31

```
ggplot(Births78, aes(x = wday, y = births)) + geom_violin(fill = "burlywood") +
  geom_jitter()
```

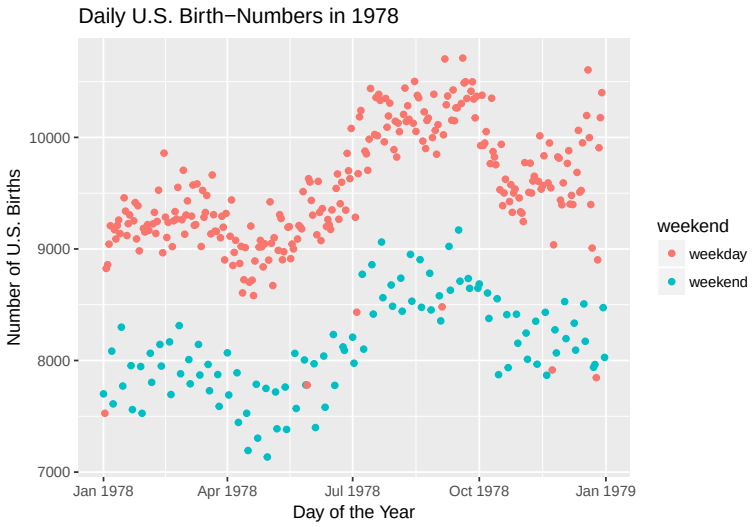


Figure 8.32: The days with fewer births are almost always weekend-days.

Aha! There are considerably fewer births on the weekend-days—Saturday and Sunday. Perhaps the *entire* lower cloud of points is composed of weekends. Let's check this by re-coding the days according to whether or not they are during the week or at the weekend:

```
weekend <- with(Births78, ifelse(wday %in% c("Sat","Sun"),
                                 "weekend", "weekday"))
Births78$weekend <- weekend
```

Note that we have added the new variable to the data frame, so that it will be easy in **ggplot2** to use that variable for grouping, as in the code below. The results appear in Figure 8.32.

```
ggplot(Births78, aes(x = date, y = births)) + geom_point(aes(color = weekend)) +
  labs(x = "Day of the Year", y = "Number of U.S. Births",
       title = "Daily U.S. Birth-Numbers in 1978")
```

Well, a *few* of the points in the lower cloud are weekdays. Is there anything special about them? To find out, we subset the data frame to examine only those points:

```
df <- subset(Births78, weekend != "weekend" & births <= 8500)
df
```

```
##           date births dayofyear  wday weekend
## 2   1978-01-02   7527         2   Mon weekday
## 149 1978-05-29   7780       149   Mon weekday
## 185 1978-07-04   8433       185  Tues weekday
## 247 1978-09-04   8481       247   Mon weekday
## 327 1978-11-23   7915       327 Thurs weekday
## 359 1978-12-25   7846       359   Mon weekday
```

If you consult a calendar for the year 1978, you will find that every one of the above days was a major holiday. Apparently doctors prefer not to deliver babies on weekend and holidays. Scheduled births—induced births or births by non-emergency Cesarean section—are not usually set for weekends or holidays. Perhaps this accounts for the two clouds we saw in the original scatter plot.

8.4 Learn More

From time to time we will return to **gplot2** and deepen our study of this remarkable graphing system. If you are impatient to learn more right way, you can explore the package's documentation site[2]. The site teaches the system by way of numerous examples that you can copy and modify.

[2] `http://docs.ggplot2.org/current/index.html`

Glossary

Frame The aesthetics (usually x and y position) that help locate cases on a plot.

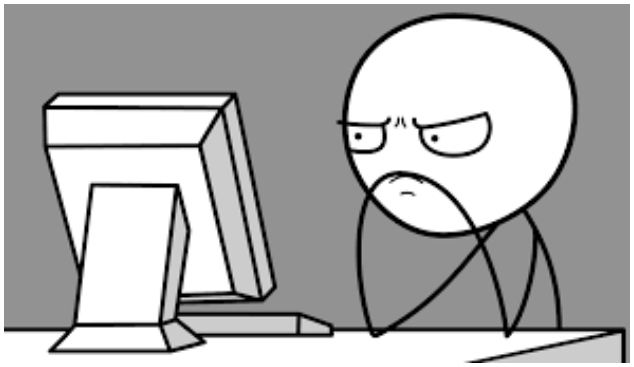
Glyph The basic graphical unit that corresponds to a case in the data table.

Aesthetic A perceptible property of a glyph that varies from case to case.

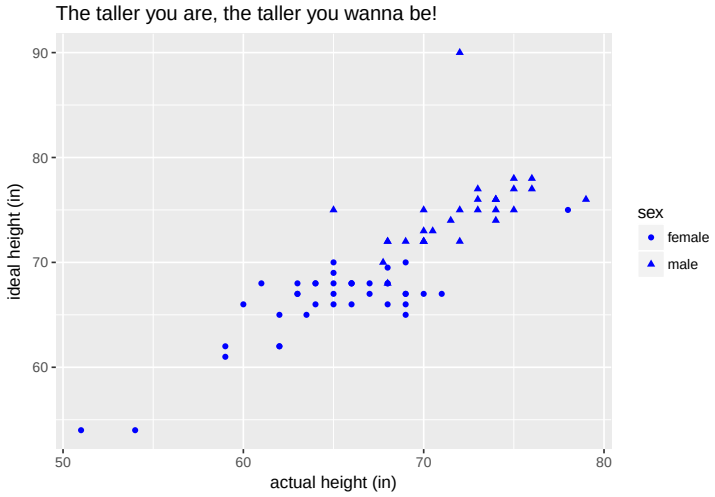
Scale The relationship between the value of a variable and the graphical attribute to be displayed for that value.

Guide An indication, for the human viewer, of the scale being used in an aesthetic mapping.

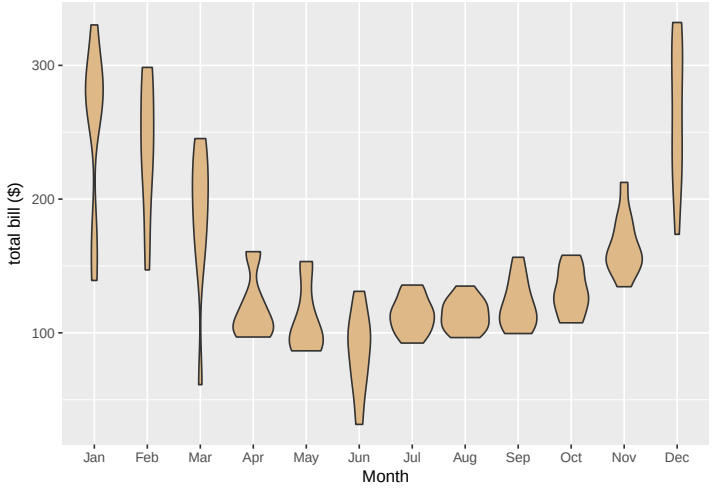
Exercises



1. Using the `tigerstats::m111survey` data frame, write the **ggplot2** code necessary to produce the graph in the figure below. (The points are all blue.)



2. Using the `mosaicData::Utilities` data frame, write the **ggplot2** code necessary to produce the graph in the figure below.



3. The next few exercises pertain to the data frame `CPS85` from the package **mosaicData**. Learn about it with `help(CPS85)`. We will use the **ggplot2** graphing package to explore whether men were being paid more than women in 1985.

Make a density plot of the wages of the people in the study. As with all plots you make, it should have well-labelled axes (with units if possible). For a density plot you should label the horizontal axis, but you can let **ggplo2** provide the label for the "density" axis. As always, provide a descriptive title. Also provide a "rug" of individual values along the horizontal axis.

4. Look at the plot you made in the previous exercise: you will notice that one person made a wage that was much higher than all the rest. In data analysis, when a value is much higher or lower than the rest of the values we call it an *outlier*.

   Write the code needed to find the age, sex and sector of employment of the person who made this extraoridinarily high wage. Report the age, sex and sector of this person.

   Create a new data frame called `cpsSmall` that is the same as `CPS85` except that it excludes the row corresponding to the outlier-individual.

5. In order to explore the relationship between wage and sex in the CPS study, make violin plots for the wages of men and women. (In this exercise and in subsequent exercises, use the `cpsSmall` data frame so as to exclude the outlier.) Based on the plot, who tends to earn higher wages: men or women?

6. Someone might argue that men don't earn higher wages because of sex-discrimination in the workplace, but rather because of some other factor. For example, it could be that in 1985 women chose to work in low-wage sectors of the economy, whereas men tended to work in higher-wage sectors. Of course for this explanation to be viable, some sectors of the economy have to pay more on average than other sectors do. In order to verify whether this is the case, make a box plot of wage vs. sector of employment. Use the plot to name a couple of high-wage sectors and a couple of low-wage sectors.

7. From the previous exercise you now know that some sectors of the economy pay more than other sectors. Hence in order to investigate properly whether there was wage-discrimination in the workforce based on sex, we would have to compare the wages of men and women who work in the *same* sector. To this end it would be nice to have eight separate box plots, one for each sector. Each plot would compare the wages of men and women in that sector. Use `facet_wrap()` to construct a graph that displays all eight plots at once.

   Examine your graph.

   - Are there any sectors in which it seems that women typically make more than men. If so, what sectors are they?
   - On the other hand, are there any sectors where men typically make more than women? If so, what sectors are they?
   - Based on your analysis, does it seem plausible that women made less than men simply because they chose lower-paying sectors of employment?

8. This exercise and the next one pertain to the data frame `imagpop` in the **tigerstats** package. Learn about it with `help(imagpop)`.

   One of the variables in `imagpop` is `kkardashtemp`, the rating given by each person to the celebrity Kim Kardashian. Make a density plot of the ratings. Compute the mean Kim Kardashian raintg for all the people in `imagpop`. Finally, compute the percentage of people in the population who gave a rating more than 40 but less than 60.

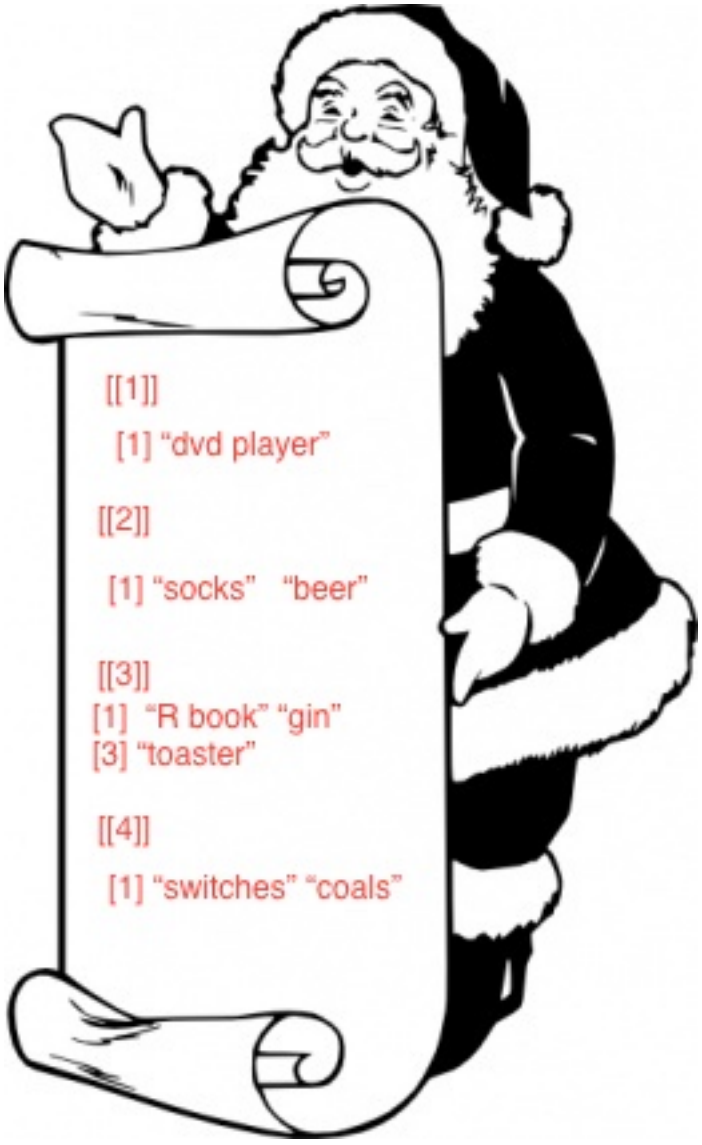
9. Write a program that repeats the following procedure 100 times:

   - Randomly select 10 people from the population.
   - Compute the mean `kkardashtemp` rating for these 10 people.

   The means should be stored in a numerical vector. Make a density plot (with rug) of the means, and also compute the percentage of the means that are between 40 and 60. As always, the plot should have sensible labels and a descriptive title.

Chapter 9

Lists



In this Chapter we will study *lists*, another important data structure in R.

9.1 Introduction to Lists

So far the vectors that we have met have all been *atomic*, meaning that they can hold only one type of value. Hence we deal with vectors of type `integer`, or of type `double`, or of type `character`, and so on.

A *list* is a special kind of vector. Like any other vector it is one-dimensional, but unlike an atomic vector it can contain objects of *any* sort: atomic vectors, functions—even other lists! We say, therefore, that lists are *heterogeneous* vectors.

The most direct way to create a list is with the function `list()` . Let's make a couple of lists:

```
lst1 <- list(name = "Dorothy", age = 12)
df <- data.frame(x = c(10, 20, 30), y = letters[1:3])
lst2 <- list(vowels = c("a", "e", "i", "o", "u"),
             myFrame = df)
lst3 <- list(nums = 10:20,
             bools = c(T, F, F),
             george = lst1)
```

Note that the elements of our three lists are not objects of a single data type. Note also that `lst3` actually contains `lst2` as one of its elements.

When you call `list()` to create a list, you have the option to assign a name to one or more of the elements. In the code above we chose, for both of our lists, to assign a name to each element.

Let's print out a list to the console. We'll choose `lst1`, since it's rather small:

```
lst1
```

```
## $name
## [1] "Dorothy"
##
## $age
## [1] 12
```

Note that the name of each elements appears before the element itself is printed out, and that the names are preceded by dollar signs. This is a hint that you can access a single member of the list in a way similar to the `frame$variable` format for data frames:

```
lst1$age
```

```
## [1] 12
```

You can make an empty list, too:

```
emptyList <- list()
```

This is useful when you want to build up a list gradually, but you do not yet know what will go into it.

9.2 Subsetting and Accessing

You can subset lists in the same way that you subset a vector: simply use the `[` sub-setting operator. Let's pick out the first two elements of `lst3`:

```
lst3[1:2]
```

```
## $nums
##  [1] 10 11 12 13 14 15 16 17 18 19 20
##
## $bools
## [1]  TRUE FALSE FALSE
```

We get a new list consisting of the desired two elements.

Suppose we want to pick out just one element from `lst3`: the numbers, for instance. We could try this:

```
justNumbers <- lst3[1]
justNumbers
```

```
## $nums
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Now suppose that we want the third number in the `nums` vector. You might think this would work fine:

```
justNumbers[3]
```

```
## $<NA>
## NULL
```

Wait a minute! The third number in `nums` is 12: so why are we getting `NA`?

Look carefully again at the printout for `justNumbers`:

```
justNumbers
```

```
## $nums
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

The `$nums` give us the clue: `justNumbers` is not just the vector `nums`—in fact it's not an atomic vector at all. It is a list whose only element is a vector with the name `nums`. Another way to see this is to check the length of `justNumbers`:

```
length(justNumbers)
```

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

The fact is that the sub-setting operator `[`, applied to lists, always returns a list. If you want access to an individual element of a list, then you need to use the double-bracket `[[` operator:

```
reallyJustNumbers <- lst3[[1]]
reallyJustNumbers
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Of course if an element of a list is named, then you may also use the dollar sign:

```
lst3$nums
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

From time to time it's useful to "flatten out" a list into a vector of values of its elements. This is accomplished by the function `unlist()` :

```
unlist(lst1)
```

```
##      name       age
## "Dorothy"      "12"
```

As the example above shows, you have to exercise caution with `unlist()`. Since `unlist()` returns an atomic vector, when it encounters values of different types then it has to coerce them to be of the same type. In the competition between `double` and `character` types, `character` wins, so you end up with a vector of strings.

9.3 Splitting

Sometimes it is useful to split a vector or data frame into pieces according to the value of a variable. For example, from `m111survey` we might like to have separate data frames for each of the three seating preferences. We can accomplish this with the `split()` function:

```
bySeat <- split(m111survey, f = m111survey$seat)
```

If you run the command `str(bySeat)`, you find that `bySeat` is a list consisting of three data frames:

- `1_front`: the frame of all subjects who prefer the Front;
- `2_middle`: the frame of all subjects who prefer the Middle;
- `3_back`: the frame of all subjects who prefer the Back.

Now you can carry on three separate analyses, working with one frame at a time.

There is a pitfall which of you should be aware. If you try to access any one of the frames by its name, you will get an error:

```
bySeat$1_front
```

```
## Error: unexpected numeric constant in "bySeat$1"
```

The reason is that variable names cannot begin with a number! You have to be content with

```
bySeat[[1]]
```

9.4 Returning Multiple Values

Lists combine many different sorts of objects into one object. This makes them very useful in the context of some functions.

Consider, for example, the drunken-turtle simulation from Section 6.8:

```
drunkenSim <- function(steps = 1000, reps = 10000, close = 0.5,
                       seed = NULL, table = FALSE) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  returns <- numeric(reps)

  for (i in 1:reps) {
  angle <- runif(steps, 0 , 2*pi)
  xSteps <- cos(angle)
  ySteps <- sin(angle)

  x <- cumsum(xSteps)
  y <- cumsum(ySteps)

  dist <- sqrt(x^2 + y^2)
  closeReturn <- (dist < 0.5)
  returns[i] <- sum(closeReturn)
  }

  if ( table ) {
    cat("Here is a table of the number of close returns:\n\n")
    tab <- prop.table(table(returns))
    print(tab)
    cat("\n")
  }
  cat("The average number of close returns was:  ",
      mean(returns), ".", sep = "")
}
```

Suppose that we would like to store several of the results of the simulation:

- the vector of the number of close returns on each repetition;
- the table made from the close-returns vector;
- the mean number of returns.

Unfortunately a function can only return *one* object.

The solution to your problem is to make a list of the three objects we want, and then return the list. We can re-write the function so as to make all output to the console optional. The function will construct the list and return it invisibly.

```
drunkenSimList <- function(steps = 1000, reps = 10000, close = 0.5,
                       seed = NULL, verbose = FALSE) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  # get the returns:
  returns <- numeric(reps)
  for (i in 1:reps) {
  angle <- runif(steps, 0 , 2*pi)
  xSteps <- cos(angle)
```

```
  ySteps <- sin(angle)

  x <- cumsum(xSteps)
  y <- cumsum(ySteps)

  dist <- sqrt(x^2 + y^2)
  closeReturn <- (dist < 0.5)
  returns[i] <- sum(closeReturn)
  }
  # compute the table and the mean:
  tableReturns <- table(returns)
  meanReturns <- mean(returns)

  # handle output to console if user wants it
  if ( verbose ) {
    cat("Here is a table of the number of close returns:\n\n")
    print(prop.table(tableReturns))
    cat("\n")
    cat("The average number of close returns was:  ",
      meanReturns, ".", sep = "")
  }

  # assemble the desired three items into a list
  # (for conveneince, name the items)
  results <- list(tableReturns = tableReturns,
                  meanReturns = meanReturns,
                  returns = returns)
  # return the list
  invisible(results)
}
```

Now we can run the function simply to acquire the simulation results for later use:

```
simResults <- drunkenSimList(seed = 3939)
```

We can use any of the results at any time and in any way we like:

```
cat("On the first ten repetitions, the number of close returns were:\n\n\t",
    simResults$returns[1:10])
```

```
## On the first ten repetitions, the number of close returns were:
##
##   0 6 4 4 2 0 2 5 2 4
```

9.5 Iterating Over a List

Lists are one-dimensional, so you can loop over them just as you would loop over a atomic vector. Sometimes this can be quite useful.

Here is a toy example. We will write a function that, when given a list of vectors, will return a vector consisting of the means of each of the vectors in the list.

```
means <- function(vecs = list(), ...) {
  n <- length(vecs)
  if ( n == 0 ) {
    stop("Need some vectors to work with!")
  }
  results <- numeric()
  for ( vec in vecs ) {
    print(vec)
    results <- c(results, mean(vec, ...))
  }
  results
}
```

```
vec1 <- 1:5
vec2 <- 1:10
vec3 <- c(1:20, NA)
means(vecs = list(vec1, vec2, vec3), na.rm = TRUE)
```

```
## [1] 1 2 3 4 5
##  [1]  1  2  3  4  5  6  7  8  9 10
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 NA
```

```
## [1]  3.0  5.5 10.5
```

Another possibility—and one that will work a bit more quickly—is to iterate over the indices of the list of vectors:

```
means2 <- function(vecs = list(), ...) {
  n <- length(vecs)
  if ( n == 0 ) {
    return(cat("Need some vectors to work with!"))
  }
  results <- numeric(n)
  for ( i in 1:n ) {
    results[i] <- mean(vecs[[i]], ...)
  }
  results
}
```

```
means2(vecs = list(vec1, vec2, vec3), na.rm = TRUE)
```

```
## [1]  3.0  5.5 10.5
```

9.6 A Note on Ellipses

The functions of the previous section contained a mysterious `...` argument in their definitions. This is known in R as the *ellipsis* argument, and it signals the possibility that one or more additional arguments may be supplied when the function is actually called.

The following function illustrates the operation of the ellipsis argument:

```
ellipisDemo <- function(...) {
  cat("I got the following arguments:\n\n")
  print(list(...))
}
ellipisDemo(x = 3, y = "cat", z = FALSE)
```

```
## I got the following arguments:
##
## $x
## [1] 3
##
## $y
## [1] "cat"
##
## $z
## [1] FALSE
```

At this point in our study of R, `...` is useful in two ways.

9.6.1 Use #1: Passing Additional Arguments to Functions "Inside"

Look again at the code for the function `means2()`:

```
means2 <- function(vecs = list(), ...) {
  n <- length(vecs)
  if ( n == 0 ) {
    return(cat("Need some vectors to work with!"))
  }
  results <- numeric(n)
  for ( i in 1:n ) {
    results[i] <- mean(vecs[[i]], ...)
  }
  results
}
```

We plan to take the mean of some vectors and therefore the `mean()` function will be used in the body of `means2()`. However we would like the user to be able to decide how `mean()` deals with `NA`-values. When we include the ellipsis argument in the definition of `means2()` we have the option to pass its contents into `mean()`, and we exercise that option in the line:

```
results[i] <- mean(vecs[[i]], ...)
```

Now we can see what happens in the call:

```
means2(vecs = list(vec1, vec2, vec3), na.rm = TRUE)
```

The ellipsis argument will consist of the argument `na.rm = TRUE`, hence the call to `mean()` inside the loop is equivalent to:

```
results[i] <- mean(vecs[[i]], na.rm = TRUE)
```

Consider, on the other hand, the call:

```
means2(vecs = list(vec1, vec2, vec3))
```

Now the ellipsis is empty. In this case the code in the loop will be equivalent to:

```
means2(vecs = list(vec1, vec2, vec3))
```

```
## [1] 3.0 5.5  NA
```

As a result, `mean()` will use the default value of `na.rm`, which is `FALSE`. For any input-vector having `NA`-values, the mean will be computed as `NA`.

9.6.2 Use #2: Permitting Any Number of Arguments

Another application of the ellipsis argument is in the writing of functions where the number of "primary" arguments is not determined in advance.

We have seen a few R-functions that can deal with any number of arguments. `cat()` is an example:

```
cat("argument one,", "argument two,", "and as many more as you like!")
```

```
## argument one, argument two, and as many more as you like!
```

With the ellipsis argument we can do this sort of thing ourselves. For example, here is a function that takes any number of vectors as arguments and determines whether the vectors are all of the same length:

```
sameLength <- function(...) {
  vecs <- list(...)
  numVecs <- length(vecs)
  if ( numVecs <= 1 ) {
    return(cat("Need two or more vectors."))
  }
  allSame <- TRUE
  len <- length(vecs[[1]])
  for ( i in 2:numVecs ) {
    if ( length(vecs[[i]]) != len ) {
      allSame <- FALSE
      break
    }
  }
  allSame
}
```

We can give this function two or more vectors, as follows:

```
vec1 <- 1:3
vec2 <- 1:4
vec3 <- 1:3
sameLength(vec1, vec2, vec3)
```

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

9.7 Investigate Your Object: `str()` and Lists

Let's reconsider the Meetup Simulation from Section 6.4:

```
meetupSim <- function(reps = 10000, table = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  anna <- runif(reps, 0, 60)
  raj <- runif(reps, 0, 60)
  connect <- (abs(anna - raj) < 10)
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    print(table(connect))
    cat("\n")
  }
  cat("The proportion of tims they met was ", mean(connect), ".\n", sep = "")
}
```

You will recall that when the user asks for a table of results, the function prints out a table that looks like this:

```
## Here is a table of the results:

## connect
## FALSE  TRUE
## 69781 30219
```

There are a couple of small irritations, here:

- The name of the table ("connect") appears in the output, even though it was a name that was given in the code internal to the function. As a name for the output-table, it's not the most descriptive choice. Besides, we really don't need a name here, because have just `cat`-ed out a sentence that introduces the table.
- The names for the columns (`FALSE` and `TRUE`) again pertain to features internal to the code of the function. The user should see more descriptive names.

In order to investigate how we might deal with these issues, let's create a small table here:

```
logicalVector <- c(rep(TRUE, 6), rep(FALSE, 4))
tab <- table(logicalVector)
tab
```

```
## logicalVector
## FALSE  TRUE
##     4     6
```

One way to deal with the column-name issues might be to isolate each table value and then repackage the values. We can access the individual table-values with sub-setting. For example, the first value is:

```
tab[1]
```

```
## FALSE
##     4
```

Hence we could grab the values, create a vector from them, and then provide names for the vector that we like. Thus:

```
results <- c(tab[1], tab[2])
names(results) <- c("did not meet", "met")
results
```

```
## did not meet          met
##            4            6
```

Another approach—and this is the more instructive and generally-useful procedure—is to begin by looking carefully at the structure of the problematic object:

```
str(tab)
```

```
##  'table' int [1:2(1d)] 4 6
##  - attr(*, "dimnames")=List of 1
##   ..$ logicalVector: chr [1:2] "FALSE" "TRUE"
```

We see that

- the table has an attribute called `dimnames`
- `dimnames` is a list of length one.
- It is a named list. The name of its only element is `logicalVector`.
- The elements of this vector are the column names for the table.

If you would like to see the `dimnames` attribute all by itself, you can access it with the `attr()` function :

```
attr(tab, which = "dimnames")  # "which" says which attribute you want!
```

```
## $logicalVector
## [1] "FALSE" "TRUE"
```

You can also use `attr()` to *set* the values of an attribute. Here, we want `dimnames` to be a list of length one that does not have a name for its sole element. The following should do the trick:

```
attr(tab, which = "dimnames") <- list(c("did not meet", "met"))
```

Let's see if this worked:

```
tab
```

```
## did not meet          met
##            4            6
```

It appears to have worked very nicely! Hence we may rewrite `meetupSim()` as follows:

```
meetupSim <- function(reps = 10000, table = FALSE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  anna <- runif(reps, 0, 60)
  raj <- runif(reps, 0, 60)
  connect <- (abs(anna - raj) < 10)
```

```
  if ( table ) {
    cat("Here is a table of the results:\n\n")
    tab <- table(connect)
    attr(tab, which = "dimnames") <- list(c("did not meet", "met"))
    print(tab)
    cat("\n")
  }
  cat("The proportion of tims they met was ", mean(connect), ".\n", sep = "")
}
```

Let's try it out:

```
meetupSim(reps = 100000, table = TRUE, seed = 3939)
```

```
## Here is a table of the results:
##
## did not meet          met
##        69781        30219
##
## The proportion of tims they met was 0.30219.
```

Much better!

The moral of the story is:

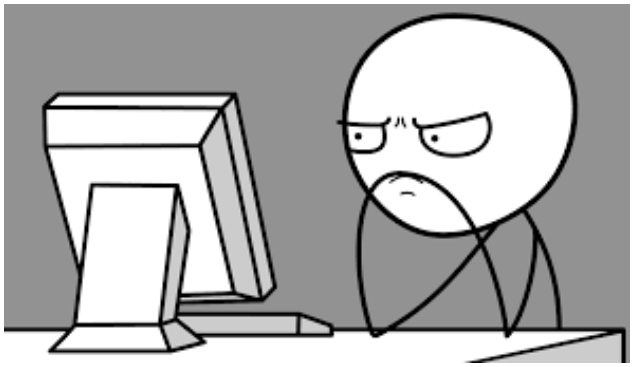
> Make a habit of examining your objects with the `str()` function. Combining `str()` with your abilities to manipulate lists allows you to access and set pieces of the object in helpful ways.

Note: the `dimnames` attribute for tables and matrices is so frequently used that it has its own special function for accessing and setting: `dimnames()`. Other popular attributes, such as `names` for a vector and `levels` for a factor, also have dedicated access/set functions—`names()` and `levels()` respectively. But keep in mind that you can access and set the values for *any attribute at all* with the `attr()` function.

Glossary

List A *heterogeneous* vector; that is, a vector whose elements can be any sort of R-object.

Exercises



1. We are given the following list:

```
lst <- list(yabba = letters,
            dabba = list(x = LETTERS,
                         y = 1:10),
            do = tigerstats::m111survey)
```

   One way to access the letter "b" in the first element of `lst` is as follows:

```
lst$yabba[2]
```

```
## [1] "b"
```

   Another way is:

```
lst[[1]][2]
```

```
## [1] "b"
```

   For each of the following objects, find at least two ways to access it within `lst`:

   - the vector of letters from "c" to "j";
   - the capital letter "F";
   - the vector of numbers from 1 to 10;
   - the heights of the five tallest individuals in `m111survey`.

2. Write a function called `goodStats()` that, when given a vector of numerical values, computes the mean, median and standard deviation of the values, and returns these values in a list. The function should take two parameters:

   - `x`: the vector of numerical values;
   - `...`: the ellipses, which allow the user to pass in additional arguments.

   The list returned should name each of the three quantities:

   - the name of the mean should be `mean`;
   - the name of the standard deviation should be `sd`;
   - the name of the median should be `median`.

   Typical examples of use should look like this:

```
vec <- 1:5
goodStats(x = vec)
```

```
## $mean
## [1] 3
##
## $sd
## [1] 1.581139
##
## $median
## [1] 3
```

```
vec <- c(3, 7, 9, 11, NA)
myStats <- goodStats(x = vec, na.rm = TRUE)
myStats$mean
```

```
## [1] 7.5
```

Chapter 10

Strings

Figure 10.1: RTL, by xkcd.

In this chapter we will take a closer look at character-vectors, and in particular at character vectors of length one, which are commonly called "strings." The ability to manipulate strings is the foundation for all text-processing in computer programming.

10.1 Character Vectors: Strings

Computers work at least as much with text as they do with numbers. In computer science the values that refer to text are called *strings*.

In R, as in most other programming languages, we use quotes as delimiters, meaning that they mark the beginning and the end of strings. Recall that in R, strings are of type `character`. For example:

```
greeting <- "hello"
typeof(greeting)
```

```
## [1] "character"
```

Of course, a single string does not exist on its own in R. Instead it exists as the only element of a character-vector of length 1.

```
is.vector(greeting)
```

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

```
length(greeting)
```

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

To make strings we can use double quotes or single quotes. Since the string-value does not include the quotes themselves but only what appears between them, it does not make any difference which type of quotes we use:

```
greeting1 <- "hello"
greeting2 <- 'hello'
greeting1 == greeting2
```

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

When we make a character vector of length greater than one, we can even use both single and double quotes:

```
politeWords <- c("Please?", 'Thank you!')
politeWords
```

```
## [1] "Please?"    "Thank you!"
```

Notice that when R prints `politeWords` to the console it uses double-quotes. Indeed, double-quoting is the recommended and most common way to construct strings in R.

10.2 Characters and Special Characters

Strings are made up of characters: that's why R calls them "character vectors." From your point of view as a speaker of the English language, characters would seem to be the things you would have entered on a typewriter, and which can be entered from your computer keyboard as well:

- the lower-case letters a-z;
- the upper case letters A-Z;

- the digits 0,1, ..., 9 (0-9);
- the punctuation characters: ., -, ?, !, ;, :, etc. (and of course the comma, too!)
- a few other special-use characters: ~, @, #, $, %, _, +, =, and so on;
- and the space, too!

All of the above can be part of a string.

But quote-marks (used in quotation and as apostrophes) can also be part of a string:

```
"Welcome", she said, "the coffee's on me!"
```

Since quote-marks are used to *delimit* strings but can also be part of them, designers of programming languages have to think carefully about how to manage quote-marks. Here's how it works in R:

- If you choose to delimit a string with double-quotes, then you can put single-quotes anywhere you like within the string and they will be treated by the computer as literal single-quotes, not as string-delimiters. Here is an example:

```
cat("'Hello', she said.")
```

```
## 'Hello', she said.
```

- If you delimit with double-quotes and you want to place a double-quote in your string, then you have to *escape* that double-quote with the backslash character \:

```
cat("\"Hello\", she said.")
```

```
## "Hello", she said.
```

- If you choose to delimit a string with single-quotes, then you can put double-quotes anywhere you like within the string and they will be treated by the computer as literal double-quotes, not as string-delimiters.

```
cat('"Hello", she said.')
```

```
## "Hello", she said.
```

- If you delimit with single-quotes and you want to place a single-quote in your string, then you have to escape that single-quote:

```
cat('\'Hello\', she said.')
```

```
## 'Hello', she said.
```

In R and in many other programming languages the backslash \ permits the following character to "escape" any special meaning that is otherwise assigned to it by the language. When we write \" we say that we are "escaping" the double-quote; more precisely, we are escaping the special role of the double-quote as a delimiter for strings.

Of course the foregoing implies that the backslash character has a special role in the language: as an escaping-device. So what can we do if we want a literal backslash in our string? Well, we simply escape it by preceding it with a backslash:

```
cat("up\\down")
```

```
## up\down
```

Table 10.1: Some control characters.

Character	Meaning
\\n	newline
\\r	carriage return
\\t	tab
\\b	backspace
\\a	alert (bell)
\\f	form feed
\\v	vertical tab

Another example:

```
cat("C:\\\\Inetpub\\\\vhosts\\\\example.com")
```

```
## C:\\Inetpub\\vhosts\\example.com
```

So much for "ordinary" characters. But there are special characters, too, sometimes called *control* characters, that do not represent written symbols. We have seen a couple of them already: the newline character `\n` is one:

```
bye <- "Farewell!\n\n"
cat(bye)
```

```
## Farewell!  # first \n moves us to a new line ...
##            # .. which is empty due the next \n
```

We have also seen the tab-character `\t`:

```
cat("First Name\tLast Name")
```

```
## First Name   Last Name
```

Notice that the backslash character is used here to allow the `n` and `t` to escape their customary roles as the letters "n" and `t` respectively.

If you ask R, (try `help(Quotes)`), you will learn that there are several control characters, including:

It is worth exploring their effects. Here are a couple of examples[1]:

```
cat("Hell\to")
```

```
## Hell o
```

```
cat("Hell\ro")
```

```
## Hell
o
```

A number of other non-control characters can be generated with the backslash. Unicode characters, for instance, are generated by `\u{nnnn}`, where the n's represent hexadecimal digits. Try the following in your console, and see what you get:

[1]Note that `cat("Hell\ao")` won't give you "Hello" with a bell-sound. To hear a bell you have to work with a terminal on your own computer. On Linux or Mac, type `echo -e "\a"` and you should hear a beep.

```
cat("\u{2603}")  # the Snowman
```

```
##
```

Or, for something zanier:

```
cat("Hello\u{202e}there, Friend!")
```

```
## Helloereht, Friend!
```

10.3 Basic String Operations

We now introduce a few basic operations for examining, splitting and combining strings.

10.3.1 Is and As

Recall from Chapter 2 that `as.character()` coerces other data types into strings:

```
as.character(3.14)
```

```
## [1] "3.14"
```

```
as.character(FALSE)
```

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

```
as.character(NULL)
```

```
## character(0)
```

Also, `is.character()` tests whether an object is character-vector:

```
is.character(3.14)
```

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

10.3.2 Number of Characters

How many characters are in the word "hello"? Let's try:

```
length("hello")
```

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

Oh, right, strings don't exist alone: `"hello"` is actually a character-vector of length 1. Instead we must use the `nchar()` function:

```
nchar("hello")
```

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

10.3.3 Substrings and Trimming

We can pull out pieces of a string with the `substr()` function:

```
poppins <- "Supercalifragilisticexpialidocious"
substr(poppins, start = 10, stop = 20)
```

```
## [1] "fragilistic"
```

One can also use `substr()` to replace part of a string with some other string[2]:

```
substr(poppins, start = 10, stop = 20) <- "ABCDEFGHIJK"
poppins
```

```
## [1] "SupercaliABCDEFGHIJKexpialidocious"
```

Don't forget: "vector-in, vector-out" usually applies:

```
words <- c("Mary", "Poppins", "practically", "perfect")
nchar(words)
```

```
## [1]  4  7 11  7
```

```
substr(words, 1, 3)
```

```
## [1] "Mar" "Pop" "pra" "per"
```

In practical data-analysis situations you'll often have to work with strings that include unexpected non-printed characters at the beginning or the end, especially if the string once occurred at the end of a line in a text file. For example, consider:

```
lastWord <- "farewell\r\n"
nchar(lastWord)
```

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

```
cat(lastWord)
```

```
## farewell
```

From its display on the console, you might infer that `lastWord` consists of only the eight characters: f, a, r, e, w, e, l, and l. (You can't see the carriage return followed by the newline.)

If you think your strings might contain unnecessary leading or trailing white-space, you can remove it with `trimws()`:

[2] In some other computer languages, strings are *immutable*, meaning that once set they cannot be changed. This is not so in R.

```
trimws(lastWord)
```

```
## [1] "farewell"
```

10.3.4 Changing Cases

You can make all of the letters in a string lowercase:

```
tolower("My name is Rhonda.")
```

```
## [1] "my name is rhonda."
```

You can make them all uppercase:

```
toupper("It makes me wanna holler!")
```

```
## [1] "IT MAKES ME WANNA HOLLER!"
```

10.3.5 Splitting Strings

Consider the following character vector that records several dates:

```
dates <- c("3-14-1963", "04-01-1965", "12-2-1983")
```

You might want to print them out in some uniform way, using the full name of the month, perhaps. Then you would need to gain access to the elements of each date separately, so that you could transform month-numbers to month-names.

`strsplit()` will do the job for you:

```
strsplit(dates, split = "-")
```

```
## [[1]]
## [1] "3"    "14"   "1963"
##
## [[2]]
## [1] "04"   "01"   "1965"
##
## [[3]]
## [1] "12"   "2"    "1983"
```

The result is a list with one element for each date in `dates`. Each element of the list is a character vector containing the elements—month-number, day-number and year—that were demarcated in the original strings by the hyphen `-`, the value given to the `split` parameter.

If we wish, we may now access the elements of the list and process them in any way we like. We might report the months, for example:

```
splitDates <- strsplit(dates, split = "-")
for ( i in 1:length(splitDates) ) {
  thisDate <- splitDates[[i]]
  thisMonth <- as.numeric(thisDate[1])
  # now use special R-constant month.names, which is a
  # character vector consisting of the full names of the months of
  # the year:
  cat(month.name[thisMonth])
  cat("\n")
}
```

```
## March
## April
## December
```

(Note the use in the code above of the `months.name` constant provided by R.)

Sometimes it's handy to split a string word-by-word:

```
message <- "you have won the lottery"
unlist(strsplit(message, split = " "))
```

```
## [1] "you"     "have"    "won"     "the"     "lottery"
```

Of course splitting on the space would not have worked if some of the words had been separated by more than one space:

```
message <- "you have won the  lottery" # two spaces betwen 'the' and 'lottery'
unlist(strsplit(message, split = " "))
```

```
## [1] "you"     "have"    "won"     "the"     ""        "lottery"
```

We'll address this problem soon.

In order to split a string into its constituent characters, split on the string with no characters:

```
animal <- "aardvark"
animSplit <- unlist(strsplit(animal, split = ""))
animSplit
```

```
## [1] "a" "a" "r" "d" "v" "a" "r" "k"
```

This would be useful if you wanted to, say, count the number of occurrences of "a" in a word:

```
length(animSplit[animSplit == "a"])
```

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

10.3.6 Pasting and Joining Strings

We are already familiar with `paste()`, which allows us to paste together the arguments that are passed to it:

```
paste("Mary", "Poppins")
```

```
## [1] "Mary Poppins"
```

By default `paste()` separates the input strings with a space, but you can control this with the `sep` parameter:

```
paste("Mary", "Poppins", sep = ":")
```

```
## [1] "Mary:Poppins"
```

```
paste("Yabba","dabba","doo!", sep = "")
```

```
## [1] "Yabbadabbadoo!"
```

If you want the separator to be the empty string by default, then you could use `paste0()`:

```
paste0("Yabba","dabba","doo!")
```

```
## [1] "Yabbadabbadoo!"
```

What if you had a character-vector whose elements you wanted to paste together? For example, consider:

```
poppins <- c("practically", "perfect", "in",
             "every", "way")
```

Now suppose you want to paste the elements of `poppins` together into one string where the words are separated by spaces. You might be tempted to try:

```
paste(poppins)
```

```
## [1] "practically" "perfect"     "in"          "every"       "way"
```

That didn't work at all. And then you remember: "Yeah, right—vector in, vector out."

There is a way, though, to get `paste()` to concatenate the elements of the resulting vector. Just use the `collapse` parameter:

```
paste(poppins, collapse = " ")
```

```
## [1] "practically perfect in every way"
```

We'll call this process *joining*.

In an atomic vector all of the elements have to be of the same data type (all character, all numerical, etc.). What if you want to join objects of different types? If there are only a few, feel free to type them in as separate arguments to `paste()`:

```
paste("March", 14, 1963, collapse = " ")
```

```
## [1] "March 14 1963"
```

If the objects are many, then you could arrange for them appear as the elements of a list:

```
toJoin <- list("Mary", 343, "Poppins", FALSE)
paste(toJoin, collapse = " ")
```

```
## [1] "Mary 343 Poppins FALSE"
```

Joining appears to be the opposite of splitting, but in R that's not quite so. Suppose, for instance, that you have dates where the month, day and year are separated by hyphens and you want to replace the hyphens with forward slashes:

```
3-14-1963  # you have this
3/14/1963  # you want this
```

You could try this:

```
date <- "3-14-1963"
splitDate <- strsplit(date, split = "-")
paste(splitDate, collapse = "/")
```

```
## [1] "c(\"3\", \"14\", \"1963\")"
```

That's not what we want. We have to remember than the result of applying `strsplit()` is a list:

```
splitDate
```

```
## [[1]]
## [1] "3"    "14"   "1963"
```

We need to unlist prior to the join. The correct procedure is:

```
date <- "3-14-1963"
splitDate <- strsplit(date, split = "-")
paste(unlist(splitDate), collapse = "/")
```

```
## [1] "3/14/1963"
```

Now all is well. Soon, though, we'll learn a superior method for performing substitutions in strings.

10.4 Formatted Printing

Quite often when we are printing out to the console we want each line to follow some uniform format. This can be accomplished with the `sprintf()` function.[3] Lets begin with an example:

```
first <- "Mary"
last <- "Poppins"
sprintf(fmt = "%10s%20s", first, last)
```

```
## [1] "      Mary             Poppins"
```

`sprintf()` builds a string from the strings `first` and `last` that were passed to it. The `fmt` parameter is a string that encodes the *format* of the result. In this example, the command comes down to:

- create a string of width 10, consisting of five spaces followed by the five characters of "Mary"

[3] You can think of `sprintf` as short for: "formatted printing in S". S was the forerunner to the R language.

- create a string of width 20, consisting of 13 spaces followed by the seven characters of "Poppins"
- The preceding two strings are called *fields*. We then join the above the fields, with nothing between them.

Here is the result, `cat`ed out:

```
cat(sprintf(fmt = "%10s%20s", first, last))
```

```
##       Mary             Poppins
```

The "s" in the the `fmt` argument is called a *conversion character*. It tells `sprintf()` to expect a string. Each percent sign indicates the beginning of a new field. For each field, the desired field-width should appear between the percent-sign and the conversion character for the field.

In the text above, the names are *right-justified*, meaning that they appear at the end of their respective fields. If you want a field to be left-justified, insert a hyphen anywhere between the percent sign and the conversion character, like so:

```
# left-justify both fields:
cat(sprintf(fmt = "%-10s%-20s", first, last))
```

```
## Mary      Poppins
```

Other common conversion characters are:

- `d`: an integer
- `f`: a decimal number (default is 6 digits precision)
- `g`: a decimal number where the default precision is determined by the number of significant figures in the given number

Here is another example:

```
cat(sprintf(fmt = "%-10s%-10d%-10f", "Mary", 1955, 3.2))
```

```
## Mary      1955      3.200000
```

The following example is the same as above, except that we retain only the significant figures in the 3.2:

```
cat(sprintf(fmt = "%-10s%-10d%-10g", "Mary", 1955, 3.2))
```

```
## Mary      1955      3.2
```

When you are creating a field for a decimal number, you can specify both the total field-width and the precision together if you separate them with a `.`. Thus, if you want the number 234.5647 to appear right-justified in a field of width 10, showing only the first three decimal places, then try:

```
cat(sprintf(fmt = "%-10s%-10d%-10.3f", "Mary", 1955, 234.5647))
```

```
## Mary      1955      234.565
```

`sprintf()` comes in handy when you want your output to appear in nicely-aligned, tabular fashion. Consider this example:

```
# information for three people:
firstName <- c("Donald", "Gina", "Rohini")
lastName <- c("Duck", "Gentorious", "Lancaster")
age <- c(17, 19, 20)
gpa <- c(3.7, 3.9, 3.823)
for (i in 1:3) {
  cat(sprintf("%-15s%-20s%-5d%-5.2f\n",
              firstName[i], lastName[i], age[i], gpa[i]))
}
```

```
## Donald         Duck                17   3.70
## Gina           Gentorious          19   3.90
## Rohini         Lancaster           20   3.82
```

Note the use of "\n" in the `fmt` argument to ensure that the output appears on separate lines.

You could take advantage of vectorization to avoid the loop:

```
cat(sprintf("%-15s%-20s%5-d%-5.2f\n",
            firstName, lastName, age, gpa))
```

```
## Donald         Duck                17   3.70
##  Gina           Gentorious          19   3.90
##  Rohini         Lancaster           20   3.82
```

Well, that's not quite right: the second and third lines begin with a space. This happens because `cat()` separates its input with a space by default. You can prevent this, however, with the `sep` parameter of `cat()`:

```
cat(sprintf("%-15s%-20s%-5d%-5.2f\n",
            firstName, lastName, age, gpa),
    sep = "")
```

```
## Donald         Duck                17   3.70
## Gina           Gentorious          19   3.90
## Rohini         Lancaster           20   3.82
```

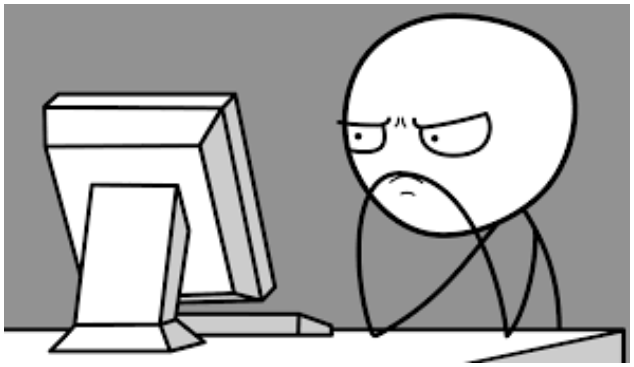
Glossary

String A sequence of characters.

Control Character A member of a character set that does not represent a written symbol.

Unicode A computing-industry standard for the consistent encoding of text in most of the world's written languages.

Exercises



1. Write a function called `revStr()` that reverses the characters of any string that it is given. The function should take a single parameter:
   - `str`: a character-vector of length 1 (a single string).

   Typical examples of use should look like this:

```
revStr(str = "goodbye")
```

```
## [1] "eybdoog"
```

2. A string is said to be a *palindrome* if it is the same no matter whether it is spelled backwards or forwards. Write a function called `palindromeStr()` that determines whether or not a given string is a palindrome. The function should take a single parameter:
   - `str`: a character-vector of length 1 (a single string).

   It should return `TRUE` if `str` is a palindrome, and return `FALSE` otherwise. Typical example of use should look like this:

```
palindromeStr(str = "abba")
```

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

```
palindromeStr("hello")
```

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

3. Write a function called `subStrings()` that returns a vector of the substrings of a given string that have at least a given number of characters. The function should take two arguments:
   - `str`: a character-vector of length 1 (a single string);
   - `n`: the minimum number of characters a substring should have in order to be includes in the vector.

   Validate the input: if the argument for `n` is less than 1 or greater than the number of characters in `str`, then the function should advise the user and cease execution. Typical examples of use should look like this (although it is OK if your output-vector contains the sub-strings in a different order):

```
subStrings("hello", 3)
```

```
## [1] "hello" "hell"  "ello"  "hel"   "ell"   "llo"
```

```
subStrings("hello", 6)
```

```
## n should be at least 1 and no more than the number
## of characters in str.
```

4. Write a function called `subPalindrome()` that, for any given string and specified number n, returns a character vector of all the substrings of the strings having at least n characters that are also palindromes. The function should take two arguments:
   - `str`: a character-vector of length 1 (a single string);
   - `n`: the minimum number of characters a substring should have in order to be included in the vector.

   Validate the input: if the argument for `n` is less than 1 or greater than the number of characters in `str`, then the function should advise the user and cease execution. Typical examples of use should look like this (although it is OK if your output-vector contains the palindromes in a different order):

```
subPalindrome("yabbadabbadoo!", 2)
```

```
##  [1] "abbadabba" "bbadabb"   "dabbad"    "badab"     "abba"
##  [6] "abba"      "ada"       "bb"        "bb"        "oo"
```

```
subPalindrome("yabbadabbadoo!", 10)
```

```
## character(0)
```

```
subPalindrome("yabbadabbadoo!", 0)
```

```
## n should be at least 1 and no more than the number
## of characters in str.
```

5. Write a function called `m111Report()` that performs formatted printing from the data frame `m111survey` in the **tigerstats** package. Given a vector of row numbers, the function will print out the sex, feeling about weight, and GPA of the corresponding individuals. Thus each row in the printout will correspond to an individual in the study. Each row will consist of three fields:
   - The first field is 10 characters wide, and contains either "male" or "female", followed by the appropriate number of spaces.
   - The first field is 15 characters wide, and contains either "underweight" or "about right" or "overweight", followed by the appropriate number of spaces.
   - The third field is 5 characters wide, and contains an appropriate number of spaces followed by the grade-point average showing only the first two decimal places. This, if a person's GPA is recorded as 2.714 then the field will be " 2.71". (Note that, with the space and the decimal point, the total number of characters is 5, as required.)

   A typical example of use is as follows:

```
m111Report(c(2, 10, 15))
```

```
## male      about right     2.50
## female    overweight        NA
## male      underweight     3.20
```

Note that you will have to re-code the feelings about weight.

Chapter 11

Regular Expressions

Figure 11.1: Regular Expressions, by xkcd.

In this Chapter we introduce the concept of *regular expressions*, a powerful tool that enables you to search for complex patterns in text.

11.1 Motivation

Suppose you wish to determine how many times the string "ab" appears within some given string. You could write a function to perform this task.

```
occurrences <- function(string) {
  count <- 0
  for (i in 1:nchar(string)) {
    stringPart <- substr(string, i, i+1)
    if ( stringPart == "ab" ) {
      count <- count +1
    }
  }
  count
}
```

Let's try it out:

```
occurrences("yabbadabbadoo!")
```

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

This looks right, as there are indeed exactly two occurrences of "ab": one at beginning at the second character and another beginning at the seventh character.

Suppose instead that we are interested in counting occurrences, in some arbitrary given string, of any of the following three strings:

- "ab"
- "Ab"
- "foo"

How might we handle this task? Again we could write a function. This time we will generalize it a bit, allowing the user to input, along with the string to be searched, a vector of the sub-strings of interest.

```
# function to count occurrences of substrings in string.
# substrings are given as patterns
occurrences2 <- function(string, patterns) {
  count <- 0
  for (i in 1:nchar(string)) {
    for (j in 1:length(patterns)) {
      pattern <- patterns[j]
      len <- nchar(pattern)
      stringPart <- substr(string, i, i + len - 1)
      if ( stringPart == pattern ) {
        count <- count +1
      }
    }
  }
  count
}
```

We try out our function on the string "This Labrador is a fool, Abba.", which matches each of our patterns exactly once, for a total of three matches,

```
occurrences2("This Labrador is a fool, Abba.",
           patterns = c("ab", "Ab", "foo"))
```

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

Well, and good, but … the coding is beginning to get a bit complex. What if we were searching instead for, say, sub-strings that resemble a phone number with an area code, i.e., strings of the form:

ddd-ddd-dddd

(Here the d's represent digits from 0 to 9.)

There are 10^{10} patterns of interest![1] How would we go about describing them all to R?

Fortunately, regular expressions are there to help us out. A *regular expression* is defined as a sequence of characters that represents a pattern that might or might not be present in any given string. A computer will rely on a *regular expression engine*—a specific implementation of a system of regular expressions—to use a given regular expression to search in text for matches to the pattern that the expression represents.

In practice, regular expressions are like a miniature programming language within a programming language. They are a feature of most major programming languages, including R. With regular expressions we can describe complex string-patterns concisely, and can perform rapid searches for these patterns in a given body of text.

The rules for regular expressions vary a bit from one language to another, but the general idea is essentially the same for all of them. In the remainder of this Chapter we'll learn enough of the principles of regular expressions to describe basic, useful patterns, and we'll also study R-functions that make use of them.

First of all, here's a quick example to show the power of regular expressions. The work done by `occurences2()` may also be done in one line with the `gregexpr()` function, as follows:

```
gregexpr(pattern = "[Aa]b|foo",
         text = "This Labrador is a fool, Abba.")
```

```
## [[1]]
## [1]  7 20 26
## attr(,"match.length")
## [1] 2 3 2
## attr(,"useBytes")
## [1] TRUE
```

We must learn to interpret output like this. The `str()` function will be helpful, here:

```
str(gregexpr(pattern = "[Aa]b|foo",
             text = "This Labrador is a fool, Abba."))
```

```
## List of 1
##  $ : atomic [1:3] 7 20 26
##   ..- attr(*, "match.length")= int [1:3] 2 3 2
##   ..- attr(*, "useBytes")= logi TRUE
```

We see that the result is a list with only one item in it. That item is a vector consisting of three numbers: 7, 20 and 26. Looking back at the text-string, we see that:

- "ab" was present beginning at position 7;
- "foo" was present beginning at position 20;

[1] 10 digits in a phone number, each of which could be chose in 10 different ways. This results in 10^{10}, or ten billion possibilities.

- "Ab" was present beginning at position 26.

The vector also has two attributes. The "useBytes" attribute will not concern us; our interest lies in the "match.length" attribute, which is yet another vector that gives the number of characters in the matching sub-strings.

R provides the convenient `regmatches()` function, which uses the beginning match position and the match lengths to fetch the matching sub-strings themselves:

```
results <- gregexpr(pattern = "[Aa]b|foo",
                    text = "This Labrador is a fool, Abba.")
regmatches("This Labrador is a fool, Abba.", m = results)
```

```
## [[1]]
## [1] "ab"  "foo" "Ab"
```

`gregexpr` is short for "global regular expression". It searches the entire`text` string for all possible matches. There is lazier function `regexpr()` that stops after the first match (if any).

What should still be mysterious is the value given to `pattern` in the call to `gregexpr()`, namely the string "[Aa]b|foo". This is the regular expression! It tells R to look for sub-strings that EITHER:

- start with either "A" or "a", and are then followed by a "b", OR
- consist of "foo"

Clearly it is high time that we learn a bit of regular-expression syntax.

11.2 Regex Practice Sites

As we introduce regex syntax in the next few sections, it's a good idea to try them out yourself and to come up with your own variations. The easiest way to do this is not to work directly in R; instead, consider using an online regex practice site. I especially recommend Regular Expressions 101[2].[3] Keep the regex flavor set at **pcre**, as this is the variant of regular expression syntax that is closest to the one implemented in R.

Bear in mind also that as we learn regex syntax, we'll focus on the standard, language-independent syntax itself. There are some differences between this standard syntax and the way in which you would actually enter a regex pattern in R.

11.3 Regex Syntax

Let's start learning the syntax.

11.3.1 Matching a Specific Sequence

If you are searching for occurrences of one specific sequence of characters, the regular expression to use is just that sequence of characters.

For example, if your regex is `bet`, then you'll get a match whenever the characters "b", "e", and "t" occur consecutively in the text you are searching.

In the sample text below, the matches are in italics:

[2] `https://regex101.com/`

[3] If you go on to learn JavaScript, you might want to switch to RegExr[4], which I think has a very nice documentation interface.

I *bet* you are reading *bet*ween the lines. Better to read the lines themselves.

We didn't match "Bet" in "Better" because "B" is uppercase.

The characters "b", "e" and "t" in the regex "bet" are examples of *literal characters*. This means that they stand for exactly what they are: the "b" in the expression matches a "b" in the text we are searching, the "e" in the expression matches an "e" in text, and so on.

There are a lot of exceptions to our specific-sequence rule. We'll get to them soon.

11.3.2 Character Classes

Suppose you want to match either "bet" or "Bet"? One way to do this is to use a *character class*. A character class consists of a set of characters surrounded by square brackets, and it tells the regex engine to match any one of the characters in the class.

Consider, for example, the regex `[Bb]et`. It consists of the character class `[Bb]` followed by the pair of literal characters `et`. It matches:

I *bet* you are reading *bet*ween the lines. *Bet*ter to read the lines themselves.

Another example: `t[aeiou]` matches any two-character sequence in which "t" is followed by a lowercase vowel:

Get thee *to* a nunnery.

11.3.2.1 Ranges

You can match a *range* of characters. Inside a character class:

- `a-z` represents all lowercase letters from a to z;
- `A-Z` represents all of the uppercase letters;
- `0-9` represents all of the decimal digits: 0, 1, 2, ..., 9.
- Other ranges are possible, e.g.:
  - `c-f` denotes the lowercase letters from c to f;
  - `0-3` denotes 9, 1, 2 and 3.

Thus, in order to match any letter followed by two digits, you could use `[a-zA-Z][0-9][0-9]`:

Your room number is *B43*, not C4 or #39.

11.3.2.2 The Need to Escape

Perhaps now you can spot the problem with the sequence-specific rule: what happens if one of the characters in the sequence is, say '[' or ']'? These characters are examples of *metacharacters*, which means that in the syntax of regular expressions they don't mach themselves but instead have a special role. In the case of square-brackets, that role is to delimit character classes.

If you want your pattern to include a metacharacter you will have to escape it with the backslash. Thus, the correct way to match the string "[aardvark]" would be with the regular expression: `\[aardvark\]`:

The *[aardvark]* appears early in the dictionary.

Actually, you only need to escape a metacharacter when it acts *in its role as metacharacter*. For example, if you want to match "b-b", you are fine to use `b-b`. Because the expression contains no square brackets, it's not possible for the hyphen to act in its special role to set ranges, so it does not have to be escaped.

Inside of square brackets it can matter whether you escape the hyphen. Thus:

Table 11.1: A few character classes worth remembering.

Class Name	Represents
[:alpha:]	a-zA-Z
[:alnum:]	a-zA-Z0-9
[:word:]	a-zA-Z0-9_ (note the underscore)
[:space:]	white space
[:lower:]	a-z
[:upper:]	A-Z

- `[a-c]` matches a, b, and c;
- `[a\-c]` matches a, -, and c (but not b);
- `[a-]` matches a and - (the machine can tell that the hyphen was not being used in a range);
- `[a\-]` also matches just a and -, not a, and -. (Apparently the regex syntax takes into account the fact that some folks will worry that they might have to escape the hyphen).

Of course since the backslash plays a role in escaping metacharacters, it too acts as a metacharacter at times. Hence if you want to match a backslash you'll have to escape *it*! How? By preceding the blackslash with a backslash! Thus:

- `a\\b` matches "a\b";
- `a\\\\b` matches "a\\b".

On the other hand:

- `a\tb` matches a followed by a tab followed by b, because in this case the machine recognizes `\t` as the control character for a tab;
- `a\ab` matches a followed by the bell-alert followed by b, and so on for other control characters.

But keep the following in mind:

- `a\eb` is incorrect regex syntax: `\e` is not recognized as one of the control characters.
- And yet `a\wb` is *correct* regex syntax! (It turns out that the token `\w` is a recognized character class *shortcut* that means the same as `[a-zA-Z0-9]`. We'll get to the shortcuts soon.)

11.3.2.3 Named Character Classes

Some character classes occur so commonly that they have been granted special names. Table 11.1 gives a few that are worth remembering.

Here's how you would use character class names in a regular expression:

- `t[[:alnum:]]t` matches t followed by any alphanumeric character followed by t. (The double brackets are needed since, according to the rules, `t[:alnum:]t` would match t followed by any one of :, a, l, n, u or m, followed by t.)

11.3.2.4 Character Class Shortcuts

Some character classes are so very common that they merit extra-short shortcuts: most of these shortcuts begin with a backslash. We'll call them *character class shortcuts*. Some of the most common character class shortcuts are shown in Table 11.2

The `.` requires special care: if you are searching for a literal dot, you'll have to escape it with a backslash. Thus `32\.456` matches "32.456", whereas `32.456` matches "32a456", 32b456", and so on.

Table 11.2: A few character class shortcuts worth remembering.

Type	Represents
\\d	any decimal digit (0-9)
\\D	anything not a decimal digit
\\s	any white space character
\\S	anything not a white space character
\\w	any word character (same as [word])
\\W	anything not a word character
.	any character except newline

11.3.2.5 Negation in a Character Class

Suppose you would like to match any sequence of three characters of this form:

- t
- any character EXCEPT e, x and z
- t

You can accomplish this by negating within a character class: the regex to use is `t[^exz]t`. Here the caret `^` functions as a metacharacter, indicating that any character except the others in the class are permitted. In order to function as a negation, the `^` must appear immediately after the opening bracket. If it appears elsewhere in the class, then it's a literal: it just stands for itself. (Outside of a character class the `^` functions as an *anchor*—we'll get to these soon—and as such has to be escaped if you want to act as a literal.)

As another example, `t[^a-z]t` matches t followed by any character expect a lowercase letter, followed by t.

Negating within a particular class of characters can be tricky. For example, suppose you are looking for 3-character sequences that consists of t, any letter except e or E, and then t. Rather than trying to use a `^` it's easiest to work with ranges: `t[a-df-zA-DF-Z]t`.

11.3.3 Quantification

Suppose you want to match phone numbers, where the area code is included and the groups of digits are separated by hyphens, as in: 202-456-1111. Taking advantage of the character class shortcut for digits, you could use the following `regex`:

```
\d\d\d-\d\d\d-\d\d\d\d
```

But that's a bit difficult to read. And what if you wanted to match sequences like the following (which has 10 digits in succession)?

```
A2356737821
```

This is where *quantifiers* come in. In regular expression syntax a token of the form `{n}` indicates that we are looking for n consecutive copies of whatever token precedes the `{n}`. Thus we can write the phone-number regex more concisely and more legibly as:

```
\d{3}-\d{3}-\d{4}
```

Note that the curly braces `{` and `}` may now function as metacharacters, and as such may have to be escaped if you are looking for them specifically, Thus if you want to search for occurrences of "t{3}b", you'll need the regex `t\{3\}b`. On the other hand if you want to match "t{swim}b" then it's fine to use the regex `t{swim}b`: the machine sees here that the braces don't play a role in quantification.

Quantifiers are quite flexible. Within a quantifier you can use a comma to indicate a range of permissible number of copies of the preceding expression:

- ‘t{3,5}1 matches 3, 4 or 5 t's in succession: ttt, tttt, or ttttt;
- `t{3,}` matches three or more t's in succession: ttt, tttt, ttttt, … .

Because quantification is so often required, regular expression syntax provides shortcuts for special cases:

- `t*` matches 0 or more t's;
- `t+` matches one or more t's;
- `t?` matches 0 or 1 t.

An example: to match the beginning of a URL, use `https?://`:

> The URL *`https://*example.org`* *provides authentication of the site, whereas you should not pay with a credit card on* `http://*flybynight.com`, On the other hand htto://example,org is not a valid URL—it's probably a typo.

Note that these shortcuts introduce new metacharacters: `*`, `+` and `?`. You'll probably need to escape them when you want to search for them as literals outside of a character class.

11.3.4 Greedy vs. Lazy

When it comes to the open-ended quantifiers (`{n,}` or the shortcuts `*` and `+`), the default behavior is to make the longest match possible. This is known as *greedy* behavior. Consider the matches reported in the following text for `b{3,}`

> b bb *bbb bbbb bbbbb*

It is possible, however, to tell a quantifier to be *lazy*, meaning that it should give the shortest possible matches. The way to do this is to append a `?` to the quantifier. Applied to the same text above the regex `b{3,}?` with the lazy quantifier reports a different set of (sometimes shorter) matches:

> b bb *bbb bbb*b *bbb*bb

11.3.5 Grouping

A quantifier refers to the shortest meaningful item immediately preceding it. Look at these examples:

- `ab+` matches “a” followed by 1 or more “b”'s;
- `a\d+` matches “a” followed by 1 or more digits;
- `a[^bc]+` matches “a” followed by 1 or more occurrence of anything other than “b” or “c”;

In the examples above, any match begins with “a”. If we want to get “a” into the scope of the quantifier, we have to group it with the item immediately preceding the quantifier. Grouping is accomplished with parentheses. Consider these examples:

- `(ab)+` matches one or more occurrences of “ab” in succession;
- `(a\d{2,})+` matches one or more occurrences of “a” followed by at least two digits. Thus it matches “a23” and “a23a773”. It won't match “a2a3”. It will match only the “a23” in “a23a6” and it will match only the “a567” in “a4a567”.

Grouping is a great help in constructing highly complex patterns. Bear in mind that the grouping symbols `(` and `)` function as metacharacters and will have to be escaped outside of character classes when you are searching for them as literals. Inside of character classes they function only as literals. Thus:

- `[(ab)*]` matches any one of the following characters: “(”, “a”, “b”, “)” and “*“.
- `\(yes\)` matches “(yes)”.

11.3.6 Alternation

In regex syntax the symbol `|` (the vertical "pipe") functions as a metacharacter meaning *or*. Thus:

- `a|b` matches "a" and it matches "b";
- `bed|bath` matches "bed" and it matches "bath";
- `a|b|c` matches any one of "a", "b" and "c";
- `(aa|bb)+` matches one or more occurrence of "aa" or "bb". Thus it matches "aa", "bb", "aaaa", "aabb", "bbaa", "bbbb" and so on.
- `(a(a|b)b)+` matches "aab", "abb", "aabaab", "aababb", and so on.

Use of the pipe-symbol is known as *alternation*.

In most implementations of regular expressions alternation works rather slowly, so use character classes instead whenever you can. For example, `[a-e]` is preferred over `a|b|c|d|e`.

11.3.7 Anchors

Sometimes we are interested in patterns that occur in specific places in text, such as:

- at the beginning of a string;
- at the end of a string
- at the beginning or end of a word.

Anchors help us accomplish this. The most important anchors to remember are:

- `^`, which indicates (in technical terms, *asserts*) the beginning of a string;
- `$`, which asserts the end of a string;[5]
- `\b`, which asserts the presence of any type word-boundary (a space, tab, comma, semicolon, etc.).

Here are some examples:

- For `^Hello`:

  Hello got matched, but not the next Hello.

- For `Hello$`:

  Hello did not get matched, but there is a match for the next *Hello*

One thing that's important to keep in mind about anchors is that they are *assertions*. This means that they don't actually count for characters in a match; they merely assert the presence of something: the beginning or end of a string, the boundary of a a word, etc. Thus, there are six tokens in the regular expression `^Hello`, but only five characters—the letters in "Hello"—are involved in a match. The`^` merely asserts that a match must not only involve the given characters but must also occur at the beginning the string.

What does `^Hello$` match? The rules would say that the string must start with H, continue on with e, l, l and then o, and end there, so you might think that the only possible string containing a match of `^Hello$` is the string "Hello" itself.

But that's not quite right:

- Go your online regex practice site (Regular Expression 101[6]).
- Enter the regex `hello$`.
- Then at the "Set Regex Options" flag (right after the "/g"), ask to enter multiline mode.
- You should now see "/gm" at the end.

[5] Well, actually it asserts the end of the string when we are not in "multiline mode." In multiline mode it asserts the end of a line within a multiline string. This will be explained shortly.

[6] `https://regex101.com/`

- In the test-text field, enter "I say hello", press Return, and continue on the next line with "Again I say hello".
- You'll see that both hello's match.

The reason for this is that in multiline mode `$` stands for the end of each line, not just the *absolute end* of the string. From time to time you may deal with strings that run over multiple lines, so remember that if you want your ending anchors to represent end-of-line rather than the absolute end of the string, you'll need to ask R to enter multiline mode. (Later on in the Chapter we'll discuss some common modes and how to enter them in R.

The word-boundary anchor `\b` is quite useful. Consider the regex `bed` applied to the string below:

> *bed bed*time perturb*bed*

There are three matches! With the regex `\bbed` there are just two matches:

> *bed bed*time perturb*bed*

With the regex `\bbed\b` the only match is with the actual word "bed":

> *bed* bedtime perturbed

Note that the beginning and the end of a string count as word-boundaries!

11.3.8 Captures

How would you detect whether a particular instance of a pattern is repeated? For instance, suppose you are looking for occurrences of a word repeated immediately after itself with only a space in between, for example:

- "*bye bye* birdie"
- "she said *night night*"

The regex `\b\w+\b` (word boundary followed by one or more word characters followed by a word boundary) will match words like "bye" and "night", but if you simply repeated the pattern, say: `\b\w+\b \b\w+\b`, then you match strings that don't exhibit repetition, such as "bye hello" and day night".

What you want is for the first part of the regex to state your pattern—a word of one or more characters—then the space, and then something that represents *exactly the match* that occurs for the first pattern.

A *capture* accomplishes this. The regex you want is:

```
\b(\w+) \1\b
```

See what it matches in the phrase below:

> now it is time for *bed bed*, yes it is bed bedtime

Here's how the regex works:

- The leading `\b` requires the presence of a word-boundary, which is satisfied by the presence, in the string, of the space between "for" and "bed bed". The second "bed bed" is also OK at this point, due to the space between "is" and the first "bed".
- `\w+` matches the first "bed", and the parentheses make it a group. By default the regex captures the contents of whatever portion of the string matches a group, and remembers those contents for later use.
- The ' ' matches the space between the two "bed" strings.
- The `\1` is a *back-reference*: it represents precisely what was matched in the earlier group. For the regex as a whole to produce a match, `\1` has to see an exact repetition of whatever string matched the first parenthesis-group in the regex, so it has to see "bed". At this point both occurrences of "bed bed"" are still in the running to be matches for the entire regex.

- The final `\b` asserts a word-boundary. This is satisfied by the comma after the first "bed bed", but not by the "t" after the second "bed bed". Thus only the first "bed bed" matches the regular expression as a whole.

Back-references are denoted `\1`, `\2`, and so on, and you can use several of them in the same regex. For example, if you want to match expressions such as "big boat big boat" then use:

```
\b(\w+) (\w+) \1 \2\b
```

Think about how the above regex works:

- To start, it requires the presence of a word-boundary.
- It sets up a capture-group consisting of one or more word characters. Since this is the first set of parentheses, the group can be referenced later on by `\1`.
- It then sets up a second capture-group that may be referenced later on by `\2`.
- We then must see a space …
- … followed by the contents of the first group …
- … followed by the contents of the second group …
- … at a word-boundary.

If you want to match a palindrome[7]consisting of five characters ("abcba", "x444x", etc.) then use:

```
\b(\w)(\w)\w\2\1\b
```

11.3.9 Looking Around

Suppose that you have a string containing a number of words involving "bed", and you would like to find all occurences of "bed" that begin a word, except for the word "bedtime". With the tools we have so far this is a difficult task. Fortunately there are *look-aheads* to simplify our work.

The regex `\bbed(?!time\b)` will do the job. Here's how it works:

- It begins by asserting a word-boundary.
- It continues with the characters to match "bed".
- It concludes a look-ahead group. The parentheses mark out the group. The initial `?` indicates that we plan to look ahead. The `!` may be thought of as "not equals"; it means that if we find the pattern that follows the `!` we will not have a match.

Note the matches in the text below:

bedtime *bed*rock *bed*rocking *bed*sheets *bed*ding embedding

Note that only "bed" is included in the match. Just like the anchor `\b`, the look-ahead is an assertion: it does not add any characters to the match.

If we want only the occurrences of "bed" where the word begins in "bed" and ends in either "rock" and "time", then we could use the regex:

```
\bbed(?=rock\b|time\b)
```

Note the matches in the following text:

*bed*time *bed*rock bedrocking bedsheets bedding embedding

Of course we could locate the same occurrences with `\bbed(rock|time)\b`, but the matches would be the entire words, not just the "bed" portion.

There are four types of look-around groups:

- *Positive look-ahead*: regex1`(?=regex2)`. Match when you find an instance of `regex2` right after an instance of `regex1`.

[7]Recall that a *palindrome* is a word that is the same when spelled backwards.

- *Negative look-ahead*: `regex1(?!regex)`. Match EXCEPT when you find an instance of `regex2` right after an instance of `regex1`.
- *Positive look-behind*: `(?<=regex2)regex1`. Match when you find an instance of `regex2` right before an instance of `regex1`.
- *Negative look-behind*: `(?<!regex2)regex1`. Match EXCEPT when you find an instance of `regex2` right before an instance of `regex1`.

The `regex2` expression in look-aheads can be any regex at all. For look-behinds, though, there are some important limitations. The precise restrictions differ from one flavor of regular expressions to another, but roughly the rule is that the machine has to be able to figure out in advance how many characters it might have to look behind. A group like `(?<=time\w*sheets)`, for instance, would not be permitted, as the quantifier `*` allows matching strings of arbitrary length.

11.3.10 More to Learn

We have not come near to exhausting the syntax of regular expressions. Readers who would like to delve into the subject more deeply should next consult online tutorials on the topics of non-capture groups, conditionals and more. However, we now have enough background to express some fairly complex patterns quite concisely, so it is now time to learn how to work with them in R.

11.4 Entering a Regex in R

We now return to the R-language and consider how to apply regular expressions within it.

11.4.1 String to Regex

Regular expressions actually play a role in one of the functions you already know, namely the function `strsplit()`. Recall that you can use the `split` parameter to specify the sub-string that separates the strings you want to split up. It works like this:

```
myString <- "hello there Mary Poppins"
# split on the spaces, then unlist:
unlist(strsplit(myString, split = " "))
```

```
## [1] "hello"   "there"   "Mary"    "Poppins"
```

The task of splitting would appear to a quite challenging if the words are separated in more complex ways, with any amount of white-space. Consider, for example:

```
myString <- "hello\t\tthere\n\nMary  \t Poppins"
cat(myString)
```

```
## hello        there
##
## Mary      Poppins
```

But really it's not any more difficult, because the `split` parameter actually takes the string it is given and converts it to a regular expression, splitting on anything that matches. Watch this:

Table 11.3: Examples of entry of regular expressions as strings, in R.

Regular Expression	Entered as String
\\s+	"\\\\s+"
find\\.dot	"find\\\\.dot "
^\\w*\\d{1,3}$	"^\\\\w*\\\\d{1,3}$"

```
myString <- "hello\t\tthere\n\nMary  \t Poppins"
unlist(strsplit(myString, split = "\\s+"))
```

```
## [1] "hello"   "there"   "Mary"    "Poppins"
```

We can almost see how this works. Recall from the last section that the regex `\s` is a character class shortcut for any white-space character, so `\s+` stands for one or more white-spaces in succession: precisely the mixtures of tab, spaces and newlines that separated the words in our string. `strsplit()` must be splitting on matches to the regex `\s+`.

So why did we set `split = "\\s+"`? What's with the extra backslash?

The reason is that the argument passed with `split` is a string, not a regular expression object. It starts out life, as if were, as a string, and R converts it to a regular expression, then hands the regex over to its regular expression engine to locate the matches in `myString` that in turn determine how `myString` is to be split up. Since in R's string-world "\s" is not a recognized character in the way that newline ("\n"), tab ("\t") and other control-characters are, R won't accept "\s+" as a valid string. Try it for your self:

```
strsplit(myString, split = "\s+")
```

```
## Error: '\s' is an unrecognized escape in character string starting ""\s"
```

It follows that when you enter regular expressions as strings in R, you'll have to remember to escape the backslashed tokens that are used in a regular expression. Table 11.3 gives several examples of this.

Keeping in mind the need for an occasional additional escape, it should not be too difficult for you to enter regular expressions in R.

11.4.2 Substitution

One of the most useful applications of regular expressions is in substitution. Suppose that we have a vector of dates:

```
dates <- c("3 - 14 - 1963", "4/13/ 2005",
           "12-1-1997", "11 / 11 / 1918")
```

It seems that the folks who entered the dates were not consistent in how to format them. In order to make analysis easier, it would be better if all the dates had exactly the same format. With the function `gsub()` and regular expressions, this is not difficult:

```
tidyDates <- gsub(pattern = "[- /]+",
                  replacement = "/",
                  x = dates)
tidyDates
```

```
## [1] "3/14/1963"  "4/13/2005"  "12/1/1997"  "11/11/1918"
```

Here:

- `pattern` is the regex that for the type of sub-string we want to replace;
- `replacment` is what we want to replace matches of the pattern with;
- `x` is the text in which the substitution occurs.

The "g" in `gsub()` stands for "global": we want to replace *all* occurrences of the pattern with the replacement text. There is also a `sub()` function that performs replacement only with the *first* match (if any) that it finds:

```
replaceFirst <- sub(pattern = "[- /]+",
                    replacement = "/",
                    x = dates)
replaceFirst
```

```
## [1] "3/14 - 1963"  "4/13/ 2005"   "12/1-1997"    "11/11 / 1918"
```

In our application, that's certainly NOT what we need. However, in cases where you happen to know that there will be at most one match, `sub()` gets the job done faster than `gsub()`, which is forced to search through the entire string.

11.4.3 A Note on Perl

R has access to a two different regular expression engines:

- POSIX 1003.2 extended regular expressions, and
- PCRE2.

POSIX 1003.2 is the default engine. "PCRE2" stands for Version 2 of the engine known as Perl-Compatible Regular Expressions.[8] It's not what R uses by default but it more closely resembles regex engines found in other major languages and is therefore the better one to use if you rely heavily on web tutorials for learning complex regular expressions. In any of R's regex functions you can access this engine with the argument:

```
perl = TRUE
```

From here on out we'll use it, simply as a matter of caution.

11.4.4 Patterned Replacement

In the dates example from Section 11.4.2 the replacement string (the argument for the parameter `replacement`) was constant: no matter what sort of match we found for the pattern `[- /]+`, we replaced it with the string "/". It is important to note, however, that if we set `perl` to `TRUE` then the argument provided for `replacement` can vary depending on the match found. In particular:

- It can include the back-references `\1`, `\2`, ..., `\9`.
- It can include special tokens for case-conversion:
  - `\U` converts the rest of the replacement to upper-case (or ceases conversion when `\E` or `\L` occur).
  - `\L` converts the rest of the replacement to lower-case (or ceases conversion when `\E` or `\U` occur).
  - `\E` signals the end of any case-conversion that is in effect.

Let's look at an example. Here is a function that, given a string, will capitalize all of the vowels that it finds:

[8] Perl is a scripting language that is known for its regular expression engine and for its text-processing capabilities in general. See the PCRE2 Syntax Page[9] for the official description of how PCRE2 works.

```
capVowels <- function(str) {
  gsub(pattern = "([aeiou])", replacement = "\\U\\1\\E", x = str, perl = TRUE)
}
capVowels("Far and away the best!")
```

```
## [1] "FAr And AwAy thE bEst!"
```

Note that the pattern `[aeiou]` for vowels had to be enclosed in parentheses so that it could be captured and referred to by the back-reference `\1`. Also note that, since R converts the replacement string into a pattern, extra backslash escapes are required, just as in regular expressions.

Here is another function that searches for repeated words and encloses each pair in asterisks:

```
starRepeats <- function(str) {
  gsub(pattern = "(\\b(\\w+) \\2\\b)", replacement = "*\\1*", x = str, perl = TRUE)
}
starRepeats("I have a boo boo on my knee knee.")
```

```
## [1] "I have a *boo boo* on my *knee knee*."
```

The above substitution is tricky, and merits step-by-step analysis:

- In order to get a repeated word, you would start with the regex

  `\b(\w+) \1\b`.

- In order to enter the regex into R as a string, you would need some escapes:

  `\\b(\\w+) \\1\\b`

- You need to wrap the whole expression in parentheses in order to capture it and refer to it in the replacement expression:

  `(\\b(\\w+) \\1\\b)`

- But now you have a problem: the back-reference `\1` applies to the *first* captured expression, which is the expression determined by the first opening parenthesis encountered. Hence it refers to the whole expression, which doesn't make sense. We want to refer to the first word in the pair (the `(\\w+)`) so we need to back-reference it with `\2`:

  `(\\b(\\w+) \\2\\b)`

- Meanwhile, in the replacement string, we want to refer to the entire repeated-word pair, so we refer to it with `\1`.

- And of course we have to do the extra backslash escaping, so the replacement string ends up as:

  `*\\1*`

Whew!

11.4.5 Grepping

If you have many strings—in a character-vector, say—and you want to know which of them contain a match to a particular pattern, then you want to use `grep()`. [10]

Consider, for example, the vector of strings:

[10] `grep` can be thought of as short for: "Global search for Regular Expression and Print".

```
sentences <- c("My name is Tom, Sir",
               "And I'm Tulip!",
               "Whereas my name is Lester.")
```

If we would like to find the strings that contain a word beginning with capital T, we could proceed as follows:

```
grep(pattern = "\\bT\\w*\\b", x = sentences, perl = TRUE)
```

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

`grep()` returns a vector consisting of the indices of the vector `sentences` where the string contains at least one word beginning with "T".

A related function is `grepl()`:

```
grepl(pattern = "\\bT\\w*\\b", x = sentences, perl = TRUE)
```

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

`grepl()` returns a logical vector with `TRUE` where `sentences` has a capital-T word, `FALSE` otherwise. We can get a look at the T-word-containing sentences themselves as follows:

```
hasTWord <- grepl(pattern = "\\bT\\w*\\b", x = sentences, perl = TRUE)
sentences[hasTWord]
```

```
## [1] "My name is Tom, Sir" "And I'm Tulip!"
```

11.4.6 More Information with `gregexpr()`

If you require more information about the matches within each string you are processing, then you'll want to use either `gregexpr()` orregexpr()‘.

We started our regular-expression journey with `gregexpr()`, but let's check in on it once more, this time looking for capital-T words in `sentences`:

```
matchInfo <- gregexpr(pattern = "\\bT\\w*\\b", text = sentences, perl = TRUE)
str(matchInfo)
```

```
## List of 3
##  $ : atomic [1:1] 12
##   ..- attr(*, "match.length")= int 3
##   ..- attr(*, "useBytes")= logi TRUE
##  $ : atomic [1:1] 9
##   ..- attr(*, "match.length")= int 5
##   ..- attr(*, "useBytes")= logi TRUE
##  $ : atomic [1:1] -1
##   ..- attr(*, "match.length")= int -1
##   ..- attr(*, "useBytes")= logi TRUE
```

From `str()` we see that `matchInfo` is a list of length three, one for each string in `sentences`. Each vector gives the starting-positions of matches in the string to which it corresponds. Since the third string contains no match the value of of the vector is set to -1.

Don't forget the convenience-function `regmatches()`, which permits us to recover the matches themselves:

```
capTNames <- regmatches(x = sentences, m = matchInfo)
capTNames
```

```
## [[1]]
## [1] "Tom"
##
## [[2]]
## [1] "Tulip"
##
## [[3]]
## character(0)
```

If we require any particular match, we can locate it easily:

```
capTNames[[2]][1]
```

```
## [1] "Tulip"
```

11.4.7 Regex Modes

If you have been practicing consistently with an online regex site, you will have noticed by now that a regex can be accompanied by various options. In most implementations these will appear as letters after the closing regex delimiter, like this:

```
/regex/gm
```

Some of the most popular options are:

- `g`: "global", looking for all possible matches in the string;
- `i`: "case-insensitive" mode, so that letter-characters in the regex match both their upper and lower-case versions;
- `m`: "multiline" mode, so that the anchors `^` and `$` are attached to newlines *within* the string rather than to the absolute beginning and end of the string;
- `x`: "whitespace" mode, where whitespaces in the regex are ignored unless they are escaped (useful for lining out the regex and inserting comments to explain its operation).

Since R has both global and non-global versions of regex functions you probably will not bother with `g`, but the various *modes* can sometimes be useful.

If you would like to set modes to apply to your entire regex, insert it (or them) like this at the beginning of the expression:

```
(?im)t[aeiou]{1,3}$
```

In the example above, we are in both case-insensitive and multiline mode, and we are looking for t or T followed by 1, 2 or 3 vowels (upper or lower) at the end of any line in a (possibly) multiline string.

Following is an example of the mode to ignore white-space:

```
myPattern <-  "(?x)        # Ignore whitespace in regex!
               \\b         # Begin the word.
               [tT]        # Upper or lower T to start,
               \\w+        # then at least one word-char,
               (?<!s)\\b   # but not ending in s.
```

```
              "
myString <- "the tars temper Tanzania"
m <- gregexpr(myPattern, myString, perl = TRUE)
regmatches(myString, m)
```

```
## [[1]]
## [1] "the"      "temper"   "Tanzania"
```

Since R knows to ignore material after `#` in any line, it does not pass the comments to the regex engine; it only passes the regex with the whitespace. Due to the presence of `(?x)` at the very beginning of the regex, the regex engine knows to ignore white-space throughout.

11.5 Application: Amazon Book Reviews

The R-package **tigerData** (White, 2017) contains the data set `reviews`, a collection of user-reviews on Amazon[11] for seven bestsellers. Learn more about the data set as follows:

```
library(tigerData)
help(reviews)
```

Each row of the data frame contains:

- the 1-5 rating that the reviewer assigned to the book
- a URL fragment that locates the review online;
- the summary-title of the review;
- the content of the review itself.

Let's focus on reviews for *Hunger Games* series. The following code creates a new data frame that contains only those reviews:

```
hunger <- subset(reviews, book == "hunger")
```

That's still a lot of reviews! We can tell by asking for the number of rows in the `hunger` data frame:

```
nrow(hunger)
```

```
## [1] 24027
```

We are looking at 24,027 reviews—some of which, by the way, are quite long.

Explore the plain-text of some of the reviews. For example, the text of the second review can be viewed with:

```
hunger$content[2]
```

```
## # A single long string.  We will show just the first few characters:
## [1] "\"<span class=\"\"a-size-base review-text\"\">Clearly ...
```

Perusing this review, we come upon the following passage:

[11] `https://www.amazon.com`

> There is a certain strain of book that can hypnotize you into believing that you are in another time and place roughly 2.3 seconds after you put that book down. <a class=\“\”a-link-normal\“\” href=””/Life-As-We-Knew-It/dp/0152061541\“\”>Life As We Knew It by Susan Beth Pfeffer could convince me that there were simply not enough canned goods in my home.

The author has linked to another book sold on Amazon, Susan Beth Pfeffer's *Life As We Knew It*. The Amazon.com URL for the book is found by prepending the company's domain to the URL-fragment seen in the excerpt above, resulting in the link:

> `http://www.amazon.com/Life-As-We-Knew-It/dp/0152061541`

We might be curious to know what other books on Amazon our reviewers link to when they are discussing the *Hunger Games*. Regular expressions can help us to extract the links from the mass of text in `hunger$content`.

All of the Amazon links are generated for the user by the computer, so they will all have the same format. Hence we can use a look-behind and a look-ahead to construct a regex will be matched by any URL-fragment within such an anchor:

> (?<=<a class=\“\”a-link-normal\“\” href=\“\”/)(.+?)(?=\“\”>)

Checking carefully, we see that none of the tokens require extra escaping: we can use this text as our pattern in regex function in R:

```
linkPattern <- "(?<=<a class=\"\"a-link-normal\"\" href=\"\"/)(.+?)(?=\"\">)"
```

We now construct a list of character-vectors containing the links (if any) in each review:

```
links <- list()
for (i in 1:nrow(hunger)) {
  text <- reviews$content[i]
  m <- gregexpr(linkPattern, text = text, perl=TRUE)
  links[[i]] <- unlist(regmatches(text, m))
}
```

If we are interested in *how many* books each reviewer links to, we can make a vector of these links:

```
numberOfLinks <- numeric()
for (i in 1:nrow(hunger)) {
  numberOfLinks[i] <- length(links[[i]])
}
```

We can tabulate the link-numbers with R's `table()` function:

```
table(numberOfLinks)
```

```
## numberOfLinks
##     0     1     2     3     4     5     6     7     8     9
## 23854   110    34    15     7     3     1     1     1     1
```

A better-looking table appears in Table 11.4.

Most of the reviewers didn't link at all, but 173 of them did provide at least one link. One reviewer linked to nine books! Here they are:

```
unlist(links[numberOfLinks == 9])
```

Table 11.4: Table showing number of links made by reviewers of the Hunger Games series.

Number of Links	Reviews
0	23854
1	110
2	34
3	15
4	7
5	3
6	1
7	1
8	1
9	1

```
## [1] "Harry-Potter-Paperback-Box-Set-Books-1-7/dp/0545162076"
## [2] "The-Dark-Tower-Boxed-Set-Books-1-4/dp/0451211243"
## [3] "The-Long-Walk/dp/0451196716"
## [4] "Battle-Royale-The-Novel/dp/1421527723"
## [5] "Battle-Royale-The-Complete-Collection-Blu-ray/dp/B006L4MX4A"
## [6] "The-Dark-Tower-Boxed-Set-Books-1-4/dp/0451211243"
## [7] "Harry-Potter-Paperback-Box-Set-Books-1-7/dp/0545162076"
## [8] "Abarat/dp/0062094106"
## [9] "lord-of-the-flies/dp/B0073SQWWC"
```

As we learn more about R's data-analysis functions we'll be able to explore a wide variety of interesting questions about the attitudes and practices of Amazon reviewers. This will involve plowing through a lot more text, but now that we know regular expressions we are sure to "save the day"!

Glossary

Regular Expression A sequence of characters that represents a pattern.

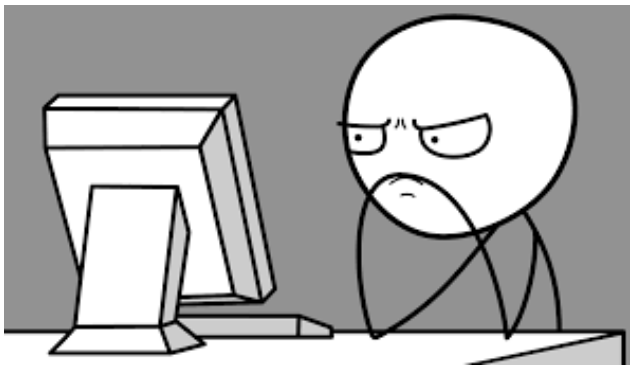
Regular Expression Engine A specific implementation of regular expressions used by a specific programming language.

Literal Character A character in a regular expression that matches itself.

Character Class A set of characters enclosed in brackets. It matches any one of the characters in the set.

Metacharacter (also called "Special Character") A character in a regular expression that does not match itself, but instead has a special role in specifying the overall pattern.

Exercises



1. Write a function called `findMister()` that, when given any string, will return a character vector of all words following the string "Mister", with exactly one space in between. The function should take a single argument called `str`, the string to search. A typical example of use is as follows:

```
text <- "Here are Mister Tom, MisterJerry, Mister Mister, and Mister\tJoe."
findMister(text)
```

```
## [1] "Tom"    "Mister"
```

2. Write a function called `findMr()` that, when given any string, will return a character vector of all words following the string "Mr.", with exactly one space in between. The function should take a single argument called `str`, the string to search. A typical example of use is as follows:

```
text <- "Here are Mr. Tom, Mr Jerry, Mr. Mister, and Mr.\tJoe."
findMr(text)
```

```
## [1] "Tom"    "Mister"
```

3. For each of the following expressions, write a regular expression to test whether any of the sub-string(s) described occur in a given string. The regular expression should match any of the sub-strings described, and should not match any other sub-string. Try to make the regular expression as short as possible. Write the regular expression as a string that could be used in one of R's regex functions (i.e. extra backslash escapes as needed). The first item is done for you, as an example.

   - *box* and *bat*. Regex string: `"b[oa]t"`. (This is the one to submit, because it's shorter than other alternatives such as`"box|bat"`).
   - *cart* and *cars* and *carp*.
   - *slick* and *sick*
   - Any word ending in *ity* (such as *velocity* and *ferocity*). Be sure to pay attention to word-boundaries. You should match *velocity* but not *velocity* (includes a space before the "v") or *velocity;*.
   - A whole number consisting of more than six digits.
   - A word that is between 3 and 6 characters long. Pay attention to word-boundaries.
   - One or more whitespace characters, followed by a hyphen or a semicolon or a colon.

4. Write a function called `findTitled()` that, when given any string, will return a character vector of all words following any one of these titles:

   - "Mr."
   - "Mister"
   - "Missus"
   - "Mrs."

- "Miss"
- "Ms."

There should be exactly one space between the title and the following word. The function should take a single argument called `str`, the string to search. A typical example of use is as follows:

```
text <- "Here are Mr. Tom, Ms. Thatcher, Miss Ellen, and Helen."
findTitled(text)
```

```
## [1] "Tom"      "Thatcher" "Ellen"
```

5. Write a function called `capRepeats()` that, when given a string, searches for all repeated-word pairs (with at least one character of whitespace in between) and replaces them with the same pair where all letters are capitalized. The function should take a single argument called `str`, the string to be searched. A typical example of use would be as follows:

```
capRepeats("I have a boo boo on my knee    \tknee!")
```

```
## [1] "I have a BOO BOO on my KNEE    \tKNEE!"
```

6. Use `grep()` or `grepl()` to write a function called `longWord()` that, when given a character vector of strings, returns a vector consisting of the strings that contain a word at least eight characters long. The function should take a single argument called `strs`. An example of use would be:

```
myText <- c("Very short words.", "Got a gargantuan word.", "More short words!")
longWord(strs = myText)
```

```
## [1] "Got a gargantuan word."
```

7. Write a function called `longWord2()` that, when given a character vector of strings, returns a list of character vectors, where each vector consists of the words in the corresponding string that are at least eight characters long. The function should take a single argument called `strs`. An example of use would be:

```
myText <- c("Very short words.", "Got a gargantuan word.", "More short words!")
longWord2(strs = myText)
```

```
## [[1]]
## character(0)
##
## [[2]]
## [1] "gargantuan"
##
## [[3]]
## character(0)
```

8. Write a function called `phoneNumber()` that, when given a vector of strings returns a logical vector indicating which of the strings contain a a valid phone number. For our purposes a valid phone number shall be any string of the form

   xxx-xxx-xxxx

   or

   xxx.xxx.xxx

Thus, 502-863-8111 is valid and so is 502.863.8111, but not 502-863.8111.

In the code for the function, specify the pattern using `(?x)` so you can ignore whitespace and leave detailed comments for each portion of the regular expression.

The function should take a single parameter called `strs`. A typical example of use would be:

```
sentences <- c("Ted's number is 606-255-3143.",
               "Rhonda's number is 403-28-1259.",
               "Lydia's number is 502.255.3921.",
               "Raj's number is 502.367-4432.")
phoneNumber(strs = sentences)
```

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

Chapter 12

Files

Figure 12.1: Scarecrow, Tin Man, Dorothy, Toto and the Cowardly Lion wearing Emerald City Specticals: an original llustration from *The Wonderful Wizard of Oz*, by L. Frank Bloom.

Most computer programming tasks involve the processing or analysis of data, and unless we are doing a simulation, that data has not been created within the program that is supposed to process it. This Chapter covers some basic techniques for importing data into R, saving it for future use, and exporting it for use by others.

12.1 Downloading Files

First we'll deal with the situation in which the data we want is in a file somewhere on the Internet. We will download the file to our computer.

It's a good idea to keep one's work organized. Since we plan to begin downloading data files, we might as well create a directory dedicated to them. It's easy to use the R Studio IDE to create the directory, but we might as well learn how the same task could be accomplished with an R-command. We will use the function `dir.create()`.

```
dir.create("downloads")
```

We are ready to begin downloading. Let's begin with a file[1] at the URL:

`https://homerhanumat.github.io/r-notes/downloads/words.zip`

The relevant function is `download.file()`:

```
download.file(url = "https://homerhanumat.github.io/r-notes/downloads/words.zip",
              destfile = "downloads/words.zip")
```

The file should now be in the `downloads` folder of your Home directory.

We see that it's a zip-file. R can unzip it for us:

```
unzip(zipfile = "downloads/words.zip")
```

We now have the text-file `words.txt` in our `downloads` directory. In R Studio you can examine the contents of the file if you click on the file-name in your **Files** tab. You see something like this:

```
aa
aah
aahed
aahing
aahs
aal
...
```

The file is a fairly comprehensive list of words in the English language—all in lowercase, one word per line.

12.2 Reading Text Files with `readLines()`

Let's transfer the contents of `words.txt` to a character vector in R. The function for this is `readLines()`:

[1] This file is a compressed version of `words.txt` file used in *Think Python* (Downey, 2015).

```
words <- readLines(con = "downloads/words.txt")
```

`readLines()` is similar to `readline()`, which we have used in the past to get a single line of input from the user. `readLines()` can also read from the console—this is called *standard input*—but it can also make a *connection*[2] with any file. In the code above, we set the connection to be the file `words.txt`. By default `readLines()` reads the file one line line at a time as a string, creating a character vector along the way. The resulting character vector is named `words`.

Let's look at words to see if it came out the way we expected. The `head()` function works not only on data frames but also on vectors:

```
head(words)
```

```
## [1] "aa"     "aah"    "aahed"  "aahing" "aahs"   "aal"
```

Yes, it looks like we got one word into each element of the vector. The total number of words is:

```
length(words)
```

```
## [1] 113809
```

12.3 Reading Text Files with readr

`words.txt` was unusually simple—just one piece of data per line. It made sense to read it into a vector. Most of the files that we encounter have a more complex structure. In many cases it is best to incorporate them into R as data frames.

R has many tools for reading data into data frames; one of the most convenient—and most popular—is the set of functions in the **readr** package (Wickham et al., 2017).

```
library(readr)
```

Let's practice reading in a file from the Internet. The URL is:

`https://homerhanumat.github.io/r-notes/downloads/alcGDP.csv`

The file is taken from data.world[3], a site very much worth exploring, and it addresses part of a study on alcohol consumption and life-expectancy in the nations of the world. (The file-extension `.csv` indicates that the it is expected to use commas to separate data values.)

The easiest way to learn the functions of **readr** is to use a widget supplied by the R Studio IDE:

- In the **Environment** Pane, find the drop-down menu **Import Dataset**,
- Ask to **Import CSV ...**.
- You will be taken to a dialog box. Enter the above URL in the **File/URL** field and press the **Update** button.
- A preview of the data is shown.
- At the bottom left there are a number of Import Options. The most important one at this point is the Name. Instead of the messy name that is shown, type something descriptive. (We'll choose `alcGDP`.)

[2] A *connection*, also known to R-programmers as a "generalized file", is any one of a wide range of interfaces that can be established with the world outside of the R-session for the purpose of sending or receiving data. One can make a connection with files stored locally on one's computer, with URL addresses on the internet, with databases, and so on.

[3] `https://data.world/jonloyens/alcohol-and-life-expectancy`

- Note the Code Preview box at the bottom right. It contains the R-commands needed to download and read the text-file `alcGDP` into R as a data frame.
- You should copy the code and save it somewhere (in an R script or R Markdown document) in order to keep a record of your work.
- You may then press the **Import** button.

The data is read into `alcGDP` according to the `read_csv()` call below:

```
alcGDP <- read_csv("https://query.data.world/s/b6plbxp3ym20s5a5iey36geul")
```

If you pressed the **Import** button, then you can see `alcGDP` in the Editor window. Notice that is has five variables. Two of them have only `NA` values, and the final two of them have names that are simply too long to be practical. Go ahead and fix this:

```
names(alcGDP)[c(4,5)] <- c("liters", "gdp")
alcGDP$YearDisplay <- NULL
alcGDP$SexDisplay <- NULL
```

Now take a look at the frame:

```
head(alcGDP)
```

```
## # A tibble: 6 x 3
##       country liters    gdp
##         <chr>  <dbl>  <int>
## 1 Afghanistan    0.0  20842
## 2     Albania    4.9  13370
## 3  Azerbaijan    1.3  75198
## 4  Madagascar    0.8  10593
## 5      Malawi    1.5   4258
## 6    Malaysia    0.3 326933
```

`liters` gives the mean total liters of alcohol consumed per person in each country, and `GDP` is the country's Gross Domestic Product in millions of dollars.

The output looks a bit different from the usual data-frame listing. That's because `alcGDP` isn't actually a data frame. `read_csv()` brought the data into R as a `tibble`, which is an object like a data frame but specially tailored for dealing with data in the context of a collection of R-packages that are commonly used by data scientists. Tibbles have many virtues, but if they make you nervous then you can get the data frame as follows:

```
alcGDP <- as.data.frame(alcGDP)
```

We won't analyze `alcGDP` here, but you might want to look at it later. You can save it permanently to your Home directory with the `save()` function:

```
save(alcGDP, file = "downloads/alcGDP.rda")
```

`alcGDP` is still in your Global Environment. If you clear the environment later on, you can reload `alcGDP` as follows:

```
load("downloads/alcGDP.rda")
```

12.4 Writing to Files

If you would like to send the data in a data frame to a colleague who does not use R, then you should convert it to a common text-file format such as CSV. For this purpose R provides the `write.csv()` command. The following code writes `m11survey` to a CSV file in your Home directory.

```
library(tigerstats)
write.csv(m111survey, file = "m111survey.csv", row.names = FALSE)
```

You may now transmit the file to your colleague.

12.5 Application: Making a Lexicon

A *lexicon* is any list of words—usually the words of a language or of some body of texts produced in that language. In this Section we'll combine our file-handling skills with regular expressions to create a lexicon for L. Frank Bloom's classic *The Wizard of Oz*.

First you need to get hold of the text itself. *The Wizrd of Oz* has been in the public domain for many years now, and is available from the Gutenberg Project[4]. You could look it up there and download it manually, but we'll practice doing it all within R. This creates a bit of a problem, since the Gutenberg Project does not permit automated file-requests, so I have installed a copy of the file at a URL that you can access:

`http://homerhanumat.github.io/r-notes/downloads/oz.txt`

Go ahead and download the file:

```
if ( !(file.exists("downloads/oz.txt")) ) {
  download.file(url = "homerhanmat.github.io/r-notes/downloads/oz.txt",
                destfile = "downloads/oz.txt")
}
```

Next, read in the file, line by line:

```
oz <- readLines(con = "downloads/oz.txt")
```

Examine `oz.txt` for a bit. If we plan to make a lexicon for the book, then we probably don't want to include Gutenberg's header material or their discussion of licensing at the end. We'll need to cut this material out of `oz`:

```
# a helper function
findIndex <- function(pattern, text) {
  inText <- grepl(pattern = pattern, x = text, perl = TRUE)
  which(inText)
}

# now find lines to start and end at:
firstLine <- findIndex("^\\*\\*\\* START OF THIS PROJECT GUTENBERG", oz) + 1
lastLine <- findIndex("^End of Project Gutenberg's", oz)  - 1
```

[4] https://www.gutenberg.org/

```
# trim oz to the desired text:
oz2 <- oz[firstLine:lastLine]
```

Let us now take a first pass at splitting the text into all of its "words." Our first thought is that words are the parts of the text that are separated by one or more white-space characters, so we might try this:

```
ozwds <- unlist(strsplit(oz2, split = "\\s+", perl = TRUE))
```

Next, let's convert all of the words to lowercase, then make a new word-list with no repeats, then sort that list:

```
ozwds <- tolower(ozwds)
ozWords <- sort(unique(ozwds))
```

It's time now to examine our prospective lexicon. Look around a bit in `ozWords`.

- There are some numbers. We'll get rid of them.
- There are an awful lot of strings that begin or end with punctuation. This should not be difficult to remove with `gsub()`.
- There are plenty of contractions, like `"aren't"`. It seems reasonable to count these as words, so we'll leave them alone.
- There are hyphenated words. What should we do with them? In a book for grown-ups you'll find lots of hyphenated words where the components words are meaningful in themselves. That's because of a grown-up grammar rule that says we should "hyphenate two or more words when they come before a noun they modify and act as a single idea." This is important: after all,"short-listed candidates" refers to candidates who appear on our short list, whereas "short listed candidates" appears to refer to short candidates who appear on our list. If this were a grown-up text then I'd want to split on hyphens in order to capture the meaningful word-components. In *The Wizard of Oz*, though, it seems that hyphenation is used mainly to create interesting new words out nonsense sound-fragments, as in:

  > So the Wicked Witch took the Golden Cap from her cupboard and placed it upon her head. Then she stood upon her left foot and said slowly:
  >
  > "Ep-pe, pep-pe, kak-ke!"
  >
  > Next she stood upon her right foot and said:
  >
  > "Hil-lo, hol-lo, hel-lo!"
  >
  > After this she stood upon both feet and cried in a loud voice:
  >
  > "Ziz-zy, zuz-zy, zik!"

  It would seem that (for example) "zuz-zy" should count as a word, but its parts "zuz" and "zy" should not count as words.
- On the other hand, we often find *pairs* of hyphens that appear to act as the *emdash* character (—):

  > The cyclone had set the house down very gently--for a cyclone--in the midst of a country of marvelous beauty.

  This causes such strings as "gently--for" and "cyclone--in" to appear in our lexicon. We don't want that, so we need to split on double-hyphens.

Let's go back and try the splitting again:

```
ozwds <- unlist(
  strsplit(oz2,
           split = "(?x)     # allow comments in regex!
                    (-{2,})  # two or more hyphens
                    |        # or
                    (\\s+)   # one or more white-space
                   ",
           perl = TRUE))
```

Now let's strip off any leading or trailing punctuation. In PCRE2, `\p{P}` is a shortcut for any punctuation character:

```
# first strip leading punctuation:
ozwds <- gsub(pattern = "^\\p{P}+",
              replacement = "",
              x = ozwds,
              perl = TRUE)
# then trailing:
ozwds <- gsub(pattern = "\\p{P}+$",
              replacement = "",
              x = ozwds,
              perl = TRUE)
# lowercase, remove duplicates, sort:
ozWords <- sort(unique(tolower(ozwds)))
```

Taking a second look at `ozWords`, we see that we need to get rid of some numbers and a spurious empty string:

```
isNumber <- grepl(pattern = "\\d+",
                  x = ozWords,
                  perl = TRUE)
isEmpty <- ozWords == ""
validWord <- !isNumber & !isEmpty
ozWords <- ozWords[validWord]
```

A final check of `ozWords` appears not turn up any serious problems. We'll take it as our lexicon for *The Wizard of Oz*.

For fun, let's make an index to the book. First, a little helper function:

```
indexFactory <- function(lexicon, fn) {
  index <- list()
  fileLines <- readLines(con = fn)
  for ( i in seq_len(length(lexicon)) ) {
    word <- lexicon[i]
    pattern <- paste0("(?i)", word)
    hasWord <- grepl(pattern = pattern, x = fileLines, perl = TRUE)
    index[[word]] <- which(hasWord)
  }
  index
}
```

Now we call our helper function to create the index:

```
ozIndex <- indexFactory(ozWords, "downloads/oz.txt")
```

We can use the index to look up words in the original text, without having to use regular expressions explicitly. First we make a convenience-function for looking up words::

```
ozLookup <- function(word) {

  fn <- "downloads/oz.txt"
  lexicon <- ozWords
  index <- ozIndex

  file <- readLines(con = fn)
  if ( !(word %in% lexicon ) ) {
    message <- paste0("\"", word, "\" is not in the lexicon!\n")
    return(cat(message))
  }
  matchLines <- index[[word]]
  number <- length(matchLines)
  cat("There are ", number, "lines that contain your request.\n\n")
  hrule <- rep("-", times = 30)
  for ( i in 1:number ) {
    lineNum <- matchLines[i]
    cat(hrule, "\n")
    cat(lineNum, ":  ", file[lineNum], "\n")
  }
}
```

Now we give it a try:

```
ozLookup("lolliop")
```

```
## "lolliop" is not in the lexicon!
```

And another try:

```
ozLookup("humbug")
```

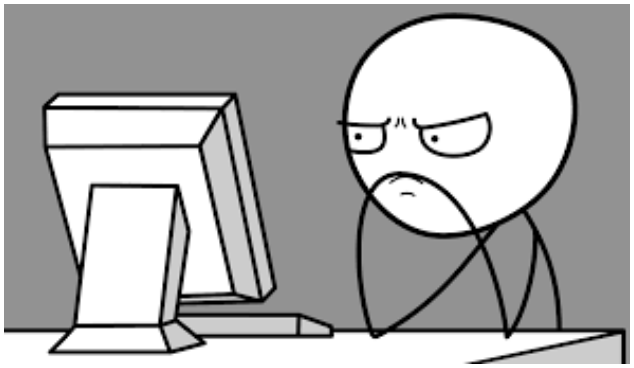
```
## There are  10 lines that contain your request.
##
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 61 :     16.  The Magic Art of the Great Humbug
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3421 :   a humbug."
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3424 :   it pleased him.  "I am a humbug."
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3438 :   "Doesn't anyone else know you're a humbug?" asked Dorothy.
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3472 :   being such a humbug."
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
## 3603 :   one I am a humbug."
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3607 :   Terrible Humbug," as she called him, would find a way to send her back
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3613 :     16.  The Magic Art of the Great Humbug
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3738 :   they wanted.  "How can I help being a humbug," he said, "when all these
## - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
## 3802 :   "Yes, of course," replied Oz.  "I am tired of being such a humbug.  If
```

Glossary

Connection (also called a "Generalized File") An interface with the world outside of the R-session, for the purpose of sending or receiving data.

Exercises



1. The word "caracara" is in `words`. Note that it in this word a sequence of four letters is immediately repeated. Write a that uses a regular expression to find all of the words in `words` that contain a doublet like this: specifically, a sequence of *four or more* letters that are immediately repeated. (The repeated sequence need NOT make up the entire word, as in the case of "caracara.") The program should `cat()` the words to the console, one per line.

2. Write a program that uses a regular expression to find all of the words in `words` that contain a "q" that is not immediately followed by a "u". The program should `cat()` the words to the console, one per line.

3. Write a program that uses a regular expression to find all of the words in `words` that are 4-letter or 5-letter palindromes. The program should `cat()` the words to the console, one per line. (**Hint:** consult Section 11.3.8.)

4. It can be proven mathematically that there is no regular expression, no matter how complex, that matches all and only the palindromes. We can write regular expressions to match all and only the palindromes that are less than some fixed number of characters, but we can't match palindromes of arbitrary length. Write a program to find all of the palindromes in `words`. Obviously this program won't have to use a regular expression!

5. Gutenburg's version of Jane Austin's classic novel *Pride and Prejudice* may be downloaded from the URL:

   `https://homerhanumat.github.io/r-notes/downloads/pride.txt`

   Your mission is to create a lexicon for Austin's novel. Follow the pattern of the work done in the text to make a lexicon for *The Wizard of Oz*. You will have to make different choices, though, about what constitutes a "word". For example, Austin's prose is complex and "grown-up"in comparison to the prose of *Oz*, so the parts of hyphenated words probably constitute valid words in and of themselves. Austin has a habit of concealing certain place-names with dashes; occasionally in letters to each other Austin's characters will refer to a person by an initial. Should initials and sequences of dashes count as words? And what about "12th" as in "the 12th of December"? As you examine the text and your initial word list you will have to make decisions about these and other matters.

   Write a report in R Markdown in which you create the lexicon. Include the code for all of your work—beginning from the file-download—so that a person who runs all of the code in the document will create the very same lexicon you made. As you develop the lexicon, explain your code and the rationale for your choices about what counts as a "word" in Austin's novel.

   Conclude the report with a data frame of the twenty most common words in Austin's novel that are more than eight letters long. The frame should have two variables: one for the words and another for the number of occurrences. The frame should be sorted so that the words appear in decreasing order of frequency (i.e., most common word comes first).

Chapter 13

Functional Programming in R

It was simple, but you know, it's always simple when you've done it.

—Simone Gabbriellini

In this Chapter we aren't going to cover any fundamentally new R powers. Instead we'll get acquainted with just one aspect of a computer programming paradigm known as *functional programming*. We will examine a set of R-functions for which functions themselves are supplied as arguments. These functions allow us to accomplish a great deal of computation in rather concise and expressive code. Not only are they useful in R itself, but they help you reason abstractly about computation and prepare you for functional programming approaches in other programming languages.

13.1 Programming Paradigms

Let us begin by exploring the notion of a *programming paradigm* in general. We will go on in this Chapter to consider will two programming paradigms for which R provides considerable support. In the next Chapter we will consider a third programming paradigm that exists in R.

A *programming paradigm* is a way to describe some of the features of programming languages. Often a paradigm includes principles concerning the use of these features, or embodies a view that these features have special importance and utility in good programming practice.

13.1.1 Procedural Programming

One of the older programming paradigms in existence is *procedural prograaming*. It is supported in many popular languages and is often the first paradigm within which beginners learn to program. In fact, if one's programming does not progress beyond a rudimentary level, one may never become aware that one is working within the procedural paradigm—or any paradigm at all, for that matter.

Before we define procedural programming, let's illustrate it with an example. Almost any of the programs we have written so far would do as examples; for specificity, let's consider the following snippet of code that produces from the data frame `m111survey` a new, smaller frame consisting of just the numerical variables:

```
library(tigerstats)
```

```
# find the numer of columns in the data frame:
cols <- length(names(m111survey))
#set up a logical vector of length equal to the number of columns:
isNumerical <- logical(cols)

# loop through.  For each variable, say if it is numerical:
for ( i in seq_along(isNumerical) ) {
  isNumerical[i] <- if (is.numeric(m111survey[, i])) TRUE else FALSE
}

# pick the numerical variables from the data frame
numsm111 <- m111survey[, isNumerical]
# have a look at the result:
str(numsm111)
```

```
## 'data.frame':    71 obs. of  7 variables:
##  $ height         : num  76 74 64 62 72 70.8 70 79 59 67 ...
##  $ ideal_ht       : num  78 76 NA 65 72 NA 72 76 61 67 ...
##  $ sleep          : num  9.5 7 9 7 8 10 4 6 7 7 ...
##  $ fastest        : int  119 110 85 100 95 100 85 160 90 90 ...
##  $ GPA            : num  3.56 2.5 3.8 3.5 3.2 3.1 3.68 2.7 2.8 NA ...
##  $ diff.ideal.act.: num  2 2 NA 3 0 NA 2 -3 2 0 ...
##  $ jitteredSpeeds : num  119.8 111.8 86.3 101.4 96.9 ...
```

By now there is nothing mysterious about the above code-snippet. What we want to become conscious of is the *approach* we have taken to the problem of selecting the numerical variables. In particular, observe that:

- We worked throughout with *data*, some of which, like `m111survey`, was given to us and some of which we created on our own to help solve the problem. For example, we created the variable `cols`. Note also the very helpful index-variable `i` in the `for`-loop. We set up the data structure `isNumerical` in order to hold a set of data (`TRUE`s and `FALSE`s).

- We relied on various *procedures* to create data and to manipulate that data in order to produce the desired result. Some of the procedures appeared as special blocks of code—most notably the `for`-loop, but also the `if-else` construct within the loop. Other procedures took the form of functions. As we know, a function encapsulates a useful procedure so that it can be easily reused in a wide variety of circumstances, without the user having to know the details of how it works. We know that `names()` will give us the vector of names of the columns of `m111survey`, that `length()` will tell us how many names, there are, that `is.numeric()` will tell us whether or not a given variable in `m111survey` is a numerical variable, and so on. The procedures embodied in these functions were written by other folks and we could examine them if we had the time and interest, but for the most part we are content simply to know how to access them.

Procedural programming is a paradigm that solves problems with programs that can be broken up into collections of variables, data structures and procedures. In this paradigm, there is a sharp distinction between variables and data structures on the one hand and procedures on the other. Variables and data structures are *data*—they are the "stuff" that a program manipulates to produce other data, other "stuff." Procedures do the manipulating, turning stuff into other stuff.

13.2 The Functional Programming Paradigm

Let us now turn to the second of the two major programming paradigms that we study in this Chapter: Functional Programming.

13.2.1 The Ubiquity of Functions in R

Let's a bit more closely at our code snippet. Notice how prominently functions figure into it, on nearly every line. In fact, *every* line calls at least one function! This might seem unbelievable: after all, consider the line below:

```
numsm111 <- m111survey[, isNumerical]
```

There don't appear to be any functions being called, here! But in fact **two** functions get called:

1. The so-called *assignment operator* `<-` is actually a function in disguise: the more official—albeit less readable—form of `variable <- value` is:

   ```
   "<-"(variable, value)
   ```

   Thus, to assign the value 3 to that variable `a` one could write:

   ```
   "<-"(a, 3)
   a   # check that a is really 3
   ```

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

2. The sub-setting operator for vectors `[`, more formally known as *extraction* (see `help(Extract)`) is also a function. The expression `m111survey[, isNumerical]` is actually the following function-call in disguise:

   ```
   "["(m111survey, isNumerical)
   ```

Indeed functions are ubiquitous in R. This is part of the significance of the following well-known remark by a developer of S, the precursor-language of R:

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call."
>
> —John Chambers

The second slogan indicates that functions are everywhere in R. It also corresponds to the first principle of the *functional programming* paradigm, namely:

> Computation is regarded as the evaluation of functions.

13.2.2 Functions as First-Class Citizens

So functions are ubiquitous in R. Another interesting thing about them is that even though they seem to be associated with procedures—after all, they make things happen—they are, nevertheless, also objects. They are data, or "stuff" if you like.

This may not seem obvious at first. But look at the following code, where you can ask what type of thing a function is:

```
typeof(is.numeric)
```

```
## [1] "builtin"
```

The so-called "primitive" functions of R—the functions written not in R but in C-code—are "built in" objects. On the other hand, consider this user-defined function:

```
f <- function(x) x+3
typeof(f)
```

```
## [1] "closure"
```

Functions other than primitive functions are objects of type "closure."[1]

If a function can be a certain type of thing, then it must be a "thing"—an object, something you can manipulate. For example, you can put functions in a list:

```
lst <- list(is.numeric, f)
lst
```

```
## [[1]]
## function (x)  .Primitive("is.numeric")
##
## [[2]]
## function (x)
## x + 3
```

[1] The term "closure" comes from the fact that every function we define in R consists of three elements: its formal arguments, its body, and a pointer to its enclosing environment. (Recall that in R the enclosing environment is the environment that was active at the time the function was defined, and that it is the second place—after the run-time environment—where R consult when looking up names during the execution of the function.) Due to the importance of its enclosing environment a function gets the name "closure."

Very importantly, you can make functions serve as argument for other functions, and functions can return other functions as their results. The following example demonstrates both of these possibilities.

```
cuber <- function(f) {
  g <- function(x) f(x)^3
  g
}
h <- cuber(abs)
h(-2)  # returns |-2|^3 = 2^3 = 8
```

```
## [1] 8
```

In fact, in R functions can be treated just like any variable. In computer programming, we say that such functions are *first-class citizens*.

Although it is not often stated as a separate principle of the functional programming paradigm it is true that in languages that provide support for functional programming, the following principle holds true:

> Functions are first-class citizens.

13.2.3 Minimize Side Effects

In the code-snippet under consideration, we note that there are two types of functions:

- functions that return a value;
- functions that provide output to the console or make a change in the Global Environment.

Example of the first type of function included:

- `length()`
- `names()`
- `seq_along()`
- `is.numeric()`
- the extraction-function `"["()`

A function that produced output to the console was `str()`.

The assignment function `"<-"()` added `cols`, `isNumerical` and `numsm111` to the Global Environment, and also made changes to `isNumerical` in the course of the `for`-loop.

Of course we have seen examples of functions that do two of these things at once, for example:

```
myFun <- function(x) {
  cat("myFun is running!\n")  # output to console
  x + 3                       # return a value
}
myFun(6)
```

```
## myFun is running!
```

```
## [1] 9
```

In computer programming, output to the console, along with changes of *state*—changes to the Global Environment or to the file structure of your computer—are called *side-effects*. Functions that only return values and do not produce side-effects are called *pure* functions.

A third principle of the functional programming paradigm is:

> Functions should be pure.

Now this principle is difficult to adhere to, and in fact if you were to adhere strictly to it in R then your programs would never "do" anything. There do exist quite practical programming languages in which all of the functions are pure—and this leads to some very interesting features such as that the order in which operations are evaluated doesn't affect what the function returns—but these "purely functional" languages manage purity by having other objects besides functions produce the necessary side-effects. In R we happily let our functions have side-effects: we certainly want to do some assignment, and print things out to the console from time to time.

One way that R does support the third principle of functional programming is that it makes it easy to avoid having your functions modify the Global Environment. To see this consider the following example:

```
addThree <- function(x) {
  heavenlyHash <- 5
  x+3  # returns this value
}
result <- addThree(10)
result
heavenlyHash
```

```
## [1] 13
## Error: object 'heavenlyHash' not found
```

This is as we expect: the variable `heavenlyHash` exists only in the run-time environment that is created in the call to `addThree()`. As soon as the function finishes execution that environment dies, and `heavenlyHash` dies long with it. In particular, it never becomes part of the Global Environment.

If you really want you functions to modify the Global Environment—or any environment other than its run-time environment, for that matter—then you have to take special measures. You could, for example, use the super-assignment operator `<<-`:

```
addThreeSideEffect <- function(x) {
  heavenlyHash <<- 5
  x+3  # returns this value
}
result <- addThreeSideEffect(10)
result
```

```
## [1] 13
```

```
heavenlyHash
```

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

The super-assignment operator looks for the name `heavenlyHash` in the parent environment of the run-time environment, If if finds `heavenlyHash` there then it changes its value to 5 and stops. Otherwise it looks in the next parent up, and so on until it reaches the Global Environment, at which point if it doesn't find a `heavenlyHash` it creates one and gives it the value. In the example above, assuming you ran the function from the console, the parent environment is the Global Environment and the function has made a change to it: a side-effect.

Except in the case of explicit assignment functions like `"<-"()`, changes made by functions to the Global Environment can be quite problematic. After all, we are used to using functions without having to look inside them to see how they do their work. Even if we once wrote the function ourselves, we may not remember how it works, so if it creates side effects we may not remember that it does, and calling them could interfere with other important work that the program is doing. (If the program already has `heavenlyHash` in the Global

Environment and the we call a function that changes it value, we could be in for big trouble.) Accordingly, R supports the third principle of functional programming to the extent of making it easy for you to avoid function calls that change your Global Environment.

13.2.4 Procedures as Higher-Order Function Calls

The last principle of the functional programming paradigms that we will state here isn't really a formal principle: it is really more an indication of the programming style that prevails in languages where functions are first-class objects and that provide other support for functional programming. The final principle is:

> As much as possible, procedures should be accomplished by function calls, In particular, loops should be replaced by calls to higher-order functions.

A higher-order function is simply a function that takes other functions as arguments. R provides a nice set of higher-order functions, many of which substitute for iterative procedures such as loops. In subsequent sections we will study the some of the most important higher-order functions, and see how they allow us to express some fairly complex procedures in a concise and readable way. You will also see how this style really blurs the distinction—so fundamental to procedural programming—between data and procedures. In functional programming, functions ARE data, and procedures are just function calls.

13.2.5 Functional Programming: A Summary

For our purposes, the principles of the functional programming paradigm are as follows:

- Computation consists in the evaluation of functions.
- Functions are first-class citizens in the language.
- Functions should only return values; they should not produce side-effects. (At the very least they should not modify the Global Environment unless they are dedicated to assignment in the first place.)
- As much as possible, procedures should be written in terms of function calls. In particular, loops should be replaced by calls to higher-order functions.

13.3 `lapply()` and Friends

In the remainder of the Chapter we will study important higher-order functions: functions that take a function as an argument and apply that function to each element of another data structure. As we have said previously, such functions often serve as alternatives to loops.

13.3.1 `lapply()`

Suppose that we want to generate five vectors, each of which consists of ten numbers randomly chosen between 0 and 1. We accomplish the task with a loop, as follows:

```
# set up a list of length 10:
lst <- vector(mode = "list", length = 5)
for ( i in 1:5 ) {
  lst[[i]] <- runif(10)
}
str(lst)
```

```
## List of 5
##  $ : num [1:10] 0.977 0.546 0.271 0.189 0.267 ...
##  $ : num [1:10] 0.7842 0.0384 0.5677 0.9629 0.5131 ...
##  $ : num [1:10] 0.967 0.184 0.268 0.477 0.263 ...
##  $ : num [1:10] 0.1517 0.9318 0.2411 0.3267 0.0647 ...
##  $ : num [1:10] 0.879 0.573 0.364 0.524 0.604 ...
```

If we wanted the vectors to have length $1, 4, 9, 16$, and 25, then we could write:

```
lst <- vector(mode = "list", length = 5)
for ( i in 1:5 ) {
  lst[[i]] <- runif(i^2)
}
str(lst)
```

```
## List of 5
##  $ : num 0.647
##  $ : num [1:4] 0.394 0.619 0.477 0.136
##  $ : num [1:9] 0.06738 0.12915 0.39312 0.00258 0.62021 ...
##  $ : num [1:16] 0.409 0.54 0.961 0.654 0.547 ...
##  $ : num [1:25] 0.96407 0.07147 0.95581 0.94798 0.00119 ...
```

In the first example, the elements in the vector `1:10` didn't matter—we wanted a vector of length ten each time—and in the second case the elements in the `1:10` did matter, in that it determined the length of the new vector produced. Of course in general we could apply `runif()` to each element of *any* vector at all, like this:

```
vec <- c(5, 7, 8, 2, 9)
lst <- vector(mode = "list", length = length(vec))
for ( i in seq_along(vec) ) {
  lst[[i]] <- runif(vec[i])
}
str(lst)
```

```
## List of 5
##  $ : num [1:5] 0.647 0.394 0.619 0.477 0.136
##  $ : num [1:7] 0.06738 0.12915 0.39312 0.00258 0.62021 ...
##  $ : num [1:8] 0.826 0.423 0.409 0.54 0.961 ...
##  $ : num [1:2] 0.1968 0.0779
##  $ : num [1:9] 0.818 0.942 0.884 0.166 0.355 ...
```

If we can apply `runif()` to each element of a vector, why not apply *an arbitrary function* to each element? That's what the function `lapply()` will do for us. The general form of `lapply()` is:

```
lapply(X, FUN, ...)
```

In the template above:

- `X` can be any vector (including the one-dimensional heterogeneous vectors known as lists);
- `FUN` is a function that is to be applied to each element of `X`. In the default operation of `lapply()`, each element of `X` becomes in turn the *first* argument of `FUN`.
- `...` consists of other arguments that are supplied as arguments for the `FUN` function, in case you have to set other parameters of the function in order to get it to perform in the way you would like.

The result is always a list.

With `lapply()` we can get the list in our second example as follows:

```
vec <- c(5, 7, 8, 2, 9)
lst <- lapply(X = vec, FUN = runif)
str(lst)
```

```
## List of 5
##  $ : num [1:5] 0.647 0.394 0.619 0.477 0.136
##  $ : num [1:7] 0.06738 0.12915 0.39312 0.00258 0.62021 ...
##  $ : num [1:8] 0.826 0.423 0.409 0.54 0.961 ...
##  $ : num [1:2] 0.1968 0.0779
##  $ : num [1:9] 0.818 0.942 0.884 0.166 0.355 ...
```

If we had wanted the random numbers to be between—say—4 and 8, then we would supply extra arguments to `runif()` as follows:

```
vec <- c(5, 7, 8, 2, 9)
lst <- lapply(X = vec, FUN = runif, min = 4, max = 8)
str(lst)
```

```
## List of 5
##  $ : num [1:5] 6.59 5.58 6.47 5.91 4.54
##  $ : num [1:7] 4.27 4.52 5.57 4.01 6.48 ...
##  $ : num [1:8] 7.3 5.69 5.64 6.16 7.84 ...
##  $ : num [1:2] 4.79 4.31
##  $ : num [1:9] 7.27 7.77 7.54 4.66 5.42 ...
```

The default behavior of `lapply()` is that the `X` vector supplies the first argument of `FUN`. However, if some `...` parameters are supplied then `X` substitutes for the first parameter that is not mentioned in `....` In the above example, the `min` and `max`parameters are the second and third parameters for `runif()` so `X` substitutes for the first parameter—the one that determines how many random numbers will be generated. Later on we'll see an example where `X` substitutes for a non-first parameter.

By the way: most people don't bother to include the names of the `X` and `FUN` parameters when using `lapply()`:

```
vec <- c(5, 7, 8, 2, 9)
lst <- lapply(vec, runif, min = 4, max = 8)
str(lst)
```

```
## List of 5
##  $ : num [1:5] 6.59 5.58 6.47 5.91 4.54
##  $ : num [1:7] 4.27 4.52 5.57 4.01 6.48 ...
##  $ : num [1:8] 7.3 5.69 5.64 6.16 7.84 ...
##  $ : num [1:2] 4.79 4.31
##  $ : num [1:9] 7.27 7.77 7.54 4.66 5.42 ...
```

If you want to use `lapply()` to get the results of our first example, in which each vector was of length 10, then you have to get around the fact that by default each element in the vector becomes the first argument of `FUN`. One way to do this is by setting `FUN` to be a function that you write on the spot yourself:

```
lst <- lapply(1:5, function(x) runif(10))
str(lst)
```

```
## List of 5
##  $ : num [1:10] 0.647 0.394 0.619 0.477 0.136 ...
##  $ : num [1:10] 0.764 0.744 0.826 0.423 0.409 ...
##  $ : num [1:10] 0.1968 0.0779 0.8184 0.9424 0.8842 ...
##  $ : num [1:10] 0.96407 0.07147 0.95581 0.94798 0.00119 ...
##  $ : num [1:10] 0.985 0.225 0.864 0.885 0.625 ...
```

In the code above, `FUN` has been set to an *anonymous* function that returns `runif(10)` no matter what value was supplied to it. In computer programming a function is called *anonymous* when it is not given a name. We could have used a regular "named" function, but we would have needed an extra line of code:

```
myfun <- function(x) runif(10)
lst <- lapply(1:5, myFun)
```

We lied a little bit when we said earlier that `X` was a vector: `X` could also be what in R is called an *expression*. We won't get into the topic of exactly what constitutes an expression (see `help(expression)` for more on the topic), but suffice to say that an expression is a combination of object-names and function calls. In particular—and this is a very common application of `lapply()`—`X` could be the name of a data frame. For example, suppose that you want to find the number of elements in each column of `mat111survery` that are `NA`. with `lapply()` this is not difficult:

```
library(tigerstats)
```

```
numberNA <- lapply(m111survey, function(x) sum(is.na(x)))
numberNA[1:2]   # show results just for the first two variables
```

```
## $height
## [1] 0
##
## $ideal_ht
## [1] 2
```

In code above, the `X` variable in the function is, in turn, each *column* of `mat111survey`. R "knows" that for a data frame that's probably what we want!

13.3.2 `sapply()`

`lapply()` always returns a list. Sometimes you would prefer that the result be something more "compact": a matrix, perhaps, or even a vector. In that case you should look into `sapply()`, which you make think of as short for "simplify-apply."

Consider the problem we brought up in the previous section: finding out how many `NA`s there are in each variable of a data frame. It would be nice to have the results in a vector rather than a list, and `sapply()` can return a vector in this case:

```
sapply(m111survey, function(x) sum(is.na(x)), simplify = "vector")
```

```
##          height        ideal_ht           sleep         fastest
##               0               2               0               0
##     weight_feel      love_first     extra_life            seat
##               0               0               0               0
##             GPA    enough_Sleep             sex diff.ideal.act.
```

```
##               1               0               0               2
##  jitteredSpeeds
##               0
```

`sapply()` resembles `lapply()` in its operation, but takes the additional parameter `simplify` that specifies what the user would like the results to be in the end. If the parameter is not set, then `sapply()` follow some internal rules to simplify the results "as much as possible." In some cases, nothing simpler than a list can be obtained, so `sapply()` will end up returning the same things as `lapply()`. As a rule it is best to specify the result you want. If `sapply()` can't deliver the desired result then it will give you an error message so you can fix things, whereas if you leave`simplify` unspecified then you'll get the unexpected results without knowing it, which could lead to bugs in your code that are very difficult to diagnose.

Earlier we mentioned that in `lapply()` the `X` parameter substitutes for the first argument of `FUN` that is not covered by the arguments supplied in `...`. The same goes for `sapply()`.

The following is an example in which—due to what's supplied in `...`, the `X` parameter goes with a non-first parameter of `FUN`.

You may recall an Exercise in Chapter 7 where you were asked to use vectorization to simulate the decisions of judges in the Appeals Court Paradox. You can `sapply()` to produce the simulated decisions, too: as follows:

```
probs <- c(0.95, 0.90, 0.90, 0.90, 0.80)
reps <- 10000
decisions <- sapply(probs, rbinom, n = reps, size = 1,
                    simplify = "matrix")
str(decisions)  # a reps-by-5 matrix
```

```
##  int [1:10000, 1:5] 1 1 1 1 1 1 1 1 1 1 ...
```

Note that for `rbinom()`:

- `n` is the first parameter;
- `size` is the second parameter.

Therefore the elements of `probs`—which is the value assigned to the `X` parameter of `sapply()`—substitute for the *third* parameter of `rbinom()`, which is indeed the `prob` parameter that controls the chance of a correct decision.

13.3.3 `tapply()`

As a motivator for learning about `tapply()`, the next member of the `apply` family, consider the question from Chapter 8 about the `m111survey` data: *Who tends to drive the fastest: people who prefer to sit in the Front, the Middle, or the Back?*

You will recall that we investigated this question—a question about the relationship between the numerical variable `fastest` and the factor-variable `seat`—with a graph, which is reproduced here as Figure 13.1. Comparing the relative positions of the violins and looking at the locations of individual values, it appears that folks who prefer to sit in the Back tend to drive faster than front and middle-sitters do, especially since there seems to be a reasonable number of back-sitters who drive quite fast (the back-sitter violin has a long upper "neck").

```
p <- ggplot(m111survey, aes(x = seat, y = fastest))
p + geom_violin(fill = "burlywood") + geom_jitter()
```

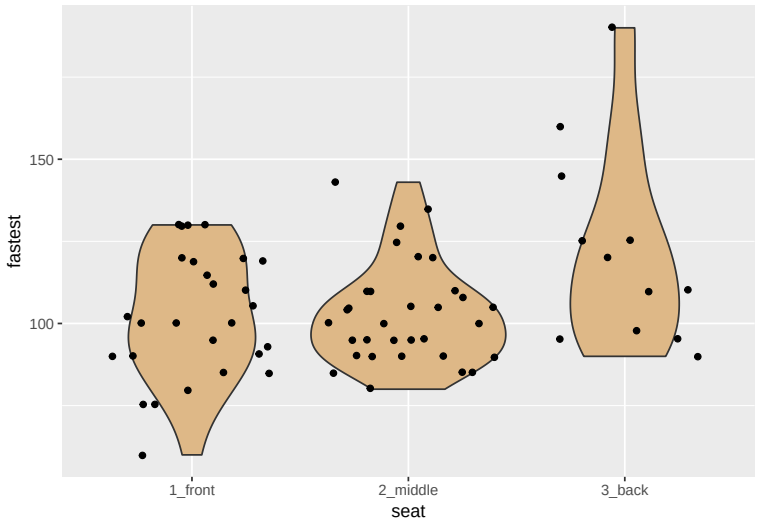


Figure 13.1: It kinds looks like the back-sitters drive faster, on the whole.

It would be nice, though, to be able to base our judgement on something other than mere eye-balling. Should we not undertake a numerical investigation of our question?

One possible numerical approach is to compare the *mean* fastest speeds of the three groups of students. We could use the `split()` function from Section 9.3 to split the vector `fastest` into a list of three vectors, one for each value of the factor `seat`. We can then compute the mean of each vector in the list and display them together for easy comparison. The following code accomplishes this:

```
df <- tigerstats::m111survey
splitSpeeds <- with(df, split(fastest, f = seat))
means <- sapply(splitSpeeds, FUN = mean, na.rm = TRUE)
means
```

```
##  1_front 2_middle   3_back
## 102.2593 102.9688 121.9167
```

Sure enough, the back-sitters have a much higher mean fastest speed!

The `tapply()` function is designed to handle these splitting situations automatically.

```
with(df, tapply(X = fastest, INDEX = seat, FUN = mean, na.rm = TRUE))
```

```
##  1_front 2_middle   3_back
## 102.2593 102.9688 121.9167
```

Here's how the interface to `tapply()` works:

- The `X` parameter `tapply()` is the vector you would like to split.
- The `INDEX` parameter gives the variable you plan to split by. It should be a factor (or something that R can coerce to a factor) and it must have the same length as `X`.
- The `FUN` parameter is the function you plan to apply to each of the vectors in the list obtained by the split.
- You can pass in other arguments that `FUN` will use. In this case we set `na.rm` to `TRUE` so that the `mean()` function will ignore missing values in any of the vectors of the split.

As it stands, `tapply()` appears to save only a line or two of code-writing. The real power of `tapply()`, however, lies in the `INDEX` parameter. `INDEX` need not be a single vector: it could be a *list* of factors, in

which case `tapply()` will split up the elements of `X` into groups determined by all possible combinations of levels of the factors in `INDEX`. The best way to understand this is with an example:

```
with(df, tapply(fastest, INDEX = list(sex, seat), FUN = mean, na.rm = TRUE))
```

```
##          1_front 2_middle   3_back
## female  99.63158  94.9375 118.0000
## male   108.50000 111.0000 124.7143
```

`tapply()` has returned a matrix with named rows and columns. (This resembles a table, hence the letter `t` in the name of the function.) Each cell of the matrix contains the mean speed for the group of people specified by the row and column names. From this "table" we can see that for each seating preference males tend to drive faster than females do, on average.

13.3.4 Map()

The apply-family of functions are quite flexible, but they are subject to the limitation that the arguments specified in `...` apply in the same way in each iteration of the `FUN`, even as the object passed in for `X` provides a *different* value to `FUN` each time. What if we would like *two or more* parameters to vary? In that case we should look to the `Map()` function.

Can you figure out what the following code does? Think about it before reading on in the text.

```
reps <- 1:5
probs <- c(0.20, 0.30, 0.50, 0.70, 0.90)
sizes <- 5:1 * 1000
sims <- Map(rbinom, n = reps, size = sizes, prob = probs)
```

Ready to read on? The code simulates what might happen if:

- there are five coins;
- the probability of Heads on each flip, for each coin, is given by the vector `probs`;
- The first coin is to be flipped 5000 times, the second coin 4000 times, the third coin 3000 times, the fourth coin 2000 times and the last coin 1000 times (as specified by `sizes`), and for each coin the number of Heads will be counted;
- For the first coin, the 5000 flips happens once, for the second coin the 4000 flips happens twice, ..., and for the fifth coin the 1000 flips happens five times. (This behavior is determined by `n = reps`.)

In `Map()` the function to be applied is the first argument. The remaining arguments will become parameters for the function. They should be vectors of the same length, but if they are not then R will attempt to recycle the shorter ones. The result of `Map()` is always a list:

```
sims
```

```
## [[1]]
## [1] 1039
##
## [[2]]
## [1] 1118 1238
##
## [[3]]
## [1] 1499 1506 1465
##
## [[4]]
```

```
## [1] 1421 1359 1393 1408
##
## [[5]]
## [1] 904 896 907 897 917
```

Here is an application of `Map()` to the simulation problem—Section 6.3—of making a triangle out of a randomly-split stick:

```
isTriangle <- function(x, y, z) {
  (x + y > z) & (x +z > y) & (y + z > x)
}

makesTriangle <- function(x, y) {
  a <- pmin(x, y)
  b <- pmax(x, y)
  side1 <- a
  side2 <- b-a
  side3 <- 1 - b
  isTriangle(x = side1, y = side2, z = side3)
}

mean(simplify2array(
  Map(makesTriangle,
      x = runif(10000),
      y = runif(10000))))  # estimate the chance of a triangle
```

```
## [1] 0.2503
```

Note the use of the function `simplify2array()` which turns the list of `TRUE`s and `FALSE`s returned by `Map()` into a logical vector. The mean of this vector gives the proportion of times in our simulation that a triangle was formed.

13.3.5 `replicate()`

`replicate()` evaluates a given expression a specified number of times. By default it simplifies the results to an array, if possible:

```
sims <- replicate(10, runif(5))
str(sims)
```

```
##  num [1:5, 1:10] 0.822 0.864 0.803 0.813 0.886 ...
```

13.4 `Filter()` and `Reduce()`

R provides support for several "higher-order" functions commonly-used in functional programming languages. (See `help(funprog)`.) Within this family of functions we have already met `Map()`. Another useful member is `Filter()`, which can do sub-setting work in situations when R's usual vector-based sub-setting falls short.

Suppose, for example, that you want to select from `mat111survey` only those variables that are factors. This can be accomplished as follows:

```
m111Factors <- Filter(is.factor, m111survey)
str(m111Factors)
```

```
## 'data.frame':    71 obs. of  6 variables:
##  $ weight_feel : Factor w/ 3 levels "1_underweight",..: 1 2 2 1 1 3 2 2 2 3 ...
##  $ love_first  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ extra_life  : Factor w/ 2 levels "no","yes": 2 2 1 1 2 1 2 2 2 1 ...
##  $ seat        : Factor w/ 3 levels "1_front","2_middle",..: 1 2 2 1 3 1 1 3 3 2 ...
##  $ enough_Sleep: Factor w/ 2 levels "no","yes": 1 1 1 1 1 2 1 2 1 2 ...
##  $ sex         : Factor w/ 2 levels "female","male": 2 2 1 1 2 2 2 2 1 1 ...
```

Note that with `Filter()` as well as with `Map()`, the function is the first argument. This is the case for all members of R's `funprog` family.

Another important member of the `funprog` family is `Reduce()` . Given a function `f` that takes two inputs and a vector `v`, `Reduce()`:

- applies `f` to elements 1 and 2 of the `v`, getting a result;
- applies `f` to the result and element 3 of `v`, getting another result;
- applies `f` to this new result and element 4 of `v`, getting yet another result …
- … and so on until all of the elements of `v` have been exhausted.
- then `Reduce()` returns the final result in the above series of operations.

For example, suppose that you want to add up the elements of the vector:

```
vec <- c(3, 1, 4, 6)
```

Of course you could just use:

```
sum(vec)
```

```
## [1] 14
```

After all, `sum()` has been written to apply to many elements at once. But what if addition could only be done two numbers at a time? How might you proceed? You could:

- add the 3 and 1 of (the first two elements of `vec`), getting 4;
- then add 4 to 4, the third element of `vec`, getting 8;
- then add 8 to 6, the final element of `vec`, getting 14;
- then return 14.

`Reduce()` operates in this way.

```
Reduce(sum, vec)
```

```
## [1] 14
```

Hence a common application of `Reduce` is to take an operation that is defined on only two items and extend it to operate on any number of items. Consider, for example, the function `intersect()`, , which will find the intersection of any two vectors of the same type:

```
vec1 <- c(3, 4, 5, 6)
vec2 <- c(4, 6, 8, -4)
intersect(vec1, vec2)
```

```
## [1] 4 6
```

You cannot intersect three or more vectors at once:

```
intersect(vec1, vec2, c(4, 7, 9))
```

```
## Error in base::intersect(x, y, ...) : unused argument (c(4, 7, 9))
```

With `Reduce()` you can intersect as many vectors as you like, provided that they are first stored in a list.

```
lst <- list(c("Akash", "Bipan", "Chandra", "Devadatta", "Raj"),
            c("Raj", "Vikram", "Sita", "Akash", "Chandra"),
            c("Akash", "Raj", "Chandra", "Bipan", "Lila"),
            c("Akash", "Vikram", "Devadata", "Raj", "Lila"))
Reduce(intersect, lst)
```

```
## [1] "Akash" "Raj"
```

13.5 Functionals vs. Loops

The functions we have studied in this chapter are often called *functionals* or *higher-order* functions because they take a function as an input. As we pointed out earlier, they deliver results that could have been produced by a writing a loop of some sort.

Once you get used to functionals, you will find that they are often more "expressive" than loops—easier for others to read and to understand, and less prone to bugs. Also, many of them are optimized by the developers of R to run a bit faster than an ordinary loop written in R.

For example, consider the following list. It consists of ten thousand vectors, each of which contains 100 randomly-generated numbers.

```
lst <- lapply(1:10000, function(x) runif(100))
```

If we want the mean of each vector, we could write a loop:

```
means <- numeric(10000)
for ( i in 1:10000 ) {
  means[i] <- mean(lst[[i]])
}
```

Or we could use `sapply()`:

```
means <- sapply(lst, mean)
```

Comparing the two using `system.time()` , on my machine I got:

```
system.time(means <- sapply(lst, mean))
```

```
##    user  system elapsed
##   1.640   0.009   1.649
```

For the loop, I get:

```
system.time({
  means <- numeric(10000)
  for ( i in 1:10000 ) {
    means[i] <- mean(lst[[i]])
  }
})
```

```
##   user  system elapsed
##  1.744   0.008   1.752
```

The apply-function is a bit faster. The difference is not too considerable, though.

Remember also that vectorization is *much* faster than looping, and is also usually quite expressive, so don't struggle to take a functional approach when vectorization is possible. (This advice applies to a number of examples from this Chapter, in which the desired computations had already been accomplished in earlier chapters by some form of vectorization.)

13.6 Conclusion

In this Chapter we have concentrated on only a single aspect of the Functional Programming paradigm: exploiting the fact that functions are first-class citizens in R, we studied a number of higher-order functions that can substitute for loops. There is certainly a great deal more to Functional Programming than the mere avoidance of loops, but we'll end our study at this point. Familiarity with higher-order functions will stand you in good stead when you begin, in subsequent courses on web programming, to learn the JavaScript language. JavaScript makes constant use of higher-order functions!

Glossary

Programming Paradigm A *programming paradigm* is a way to describe some of the features of programming languages. Often a paradigm includes principles concerning the use of these features, or embodies a view that these features have special importance and utility in good programming practice.

Procedural Programming A programming paradigm that solves problems with programs that can be broken up into collections of variables, data structures and procedures. This paradigm, tends to draw a sharp distinction between variables and data structures on the one hand and procedures on the other.

Functional Programming A programming paradigm that stresses the central role of functions. Some of its basic principles are:

- Computation consists in the evaluation of functions.
- Functions are first-class citizens in the language.
- Functions should only return values; they should not produce side-effects.
- As much as possible, procedures should be written in terms of function calls.

Pure Function A function that does not produce side-effects.

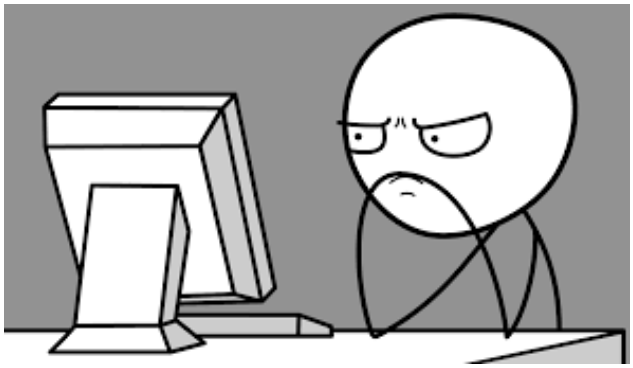
Side Effect A change in the state of the program (i.e., a change in the Global Environment) or any interaction external to the program (i.e., printing to the console).

Higher-Order Function A function that takes another function as an argument.

Anonymous Function A function that does not have a name.

Refactoring The act of rewriting computer code so that it performs the same task as before, but in a different way. (This is usually done to make the code more human-readable or to make it perform the task more quickly.)

Exercises



1. Use `tapply()` to compare the median fastest speeds of the front-sitters, middle-sitters and back-sitters in the `m111survey` data from the **tigerstats** package. Which group has the highest median?

2. The 25th percentile of a numerical vector `x` is a number such that about 25% of the values in `x` are less than that number. This number is called the *first quartile* of `x`. The 75th percentile of `x`, also called the *third quartile*, is a number so that about 75% of the values of `x` are less than that number.

   It follows that about 50% of the values of `x` lie between its first and third quartiles. The difference between the two quartiles (third quartile minus first quartile) is called the *interquartile range* (IQR) of `x`. It's a measure of how "spread out" typical values of `x` are: the bigger the IQR, the more the spread.

   Study the function `quantile()` from the **stats** package. Notice that you can get the first and third quartiles of `x` with the command

   ```
   quantile(x, probs = c(0.25, 0.75))
   ```

   Your task is to use `quantile()` to write a function called `iqRange()` that will return the interquartile range of any given numerical vector. Then use that function with `tapply()` to get the interquartile ranges for the fastest speeds in the `m111survey` data, where the students are broken into groups according to both seating preference and sex.

3. This exercise continues the study of sex discrimination in wages that was begun in the Exercises from Chapter 7. In those exercises our approach was graphical, but we would like to check our conclusions with a numerical approach as well. Working from the `cpsSmall` data frame, use `tappply()` to compare mean wages when the subjects are broken into groups according to both sex and sector of employment. (This corresponds to the graph with eight violin-plots from the Chapter 7 Exercises.)

   Examine the "table" you got with `tapply()`.

   - Are there any sectors in which it seems that women typically make more than men. If so, what sectors are they?
   - On the other hand, are there any sectors where men typically make more than women? If so, what sectors are they?
   - Based on your analysis, does it seem plausible that women made less than men simply because they chose lower-paying sectors of employment?

4. Explain in words what the following line of code produces when given a vector `y`:

   ```
   Map(function(x) x^2, y)
   ```

   In the course of your explanation, say whether the result is a vector or a list.

5. Which do you think works faster for a given numerical vector `y`? This code:

```
Map(function(x) sqrt(x), y)
```

   Or this code?

```
  sqrt(y)
```

   Justify your answer with a convincing example, using `system.time()`. What moral do you draw from this?

6. To *refactor* computer code is to rewrite the code so that is does the same thing, but in a different way. We might refactor code in order to make it more readable by humans, or to make it perform its task more quickly.

   Refactor the following code so that it uses `Map()` instead of a loop:

```
df <- tigerstats::m111survey
keepVariable <- logical(length(names(df)))
for ( col in seq_along(keepVariable) ) {
  var <- df[, col]
  isNumeric <- is.numeric(var)
  allThere <- !any(is.na(var))
  keepVariable[col] <- isNumeric && allThere
}
newFrame <- df[, keepVariable]
head(newFrame)
```

7. The following function produces a list of vectors of uniform random numbers, where the lower and upper bounds of the numbers are given by the arguments to the parameters `lower` and `upper` respectively, and the number of vectors in the list and the number of random numbers in each vector are given by a vector supplied to the parameter `vecs`.

```
randomSims <- function(vecs, lower = 0, upper= 1, seed = NULL) {
  # set seed if none is provided by the user
  if ( is.null(seed) ) {
    seed <- as.numeric(Sys.time())
  }
  set.seed(seed)

  lst <- vector(mode = "list", length = length(vecs))
  for ( i in seq_along(vecs) ) {
    lst[[i]] <- runif(vecs[i], min = lower, max = upper)
  }
  lst
}
```

   Refactor the code for `randomSims()` so that it uses `lapply()` instead of a loop.

8. The following enhanced version of `randomSims()` is even more flexible, as it allows both the upper and lower limits for the randomly-generated numbers to vary with each vector of numbers that is produced.

```
randomSims2 <- function(vecs, lower = 0, upper= 1, seed = NULL) {
  # validate input
  if (!(length(vecs) == length(upper) && length(upper) == length(lower)) ) {
    return(cat("All vectors entered must have the same length."))
  }
  if ( any(upper < lower) ) {
    return(cat(paste0("Every upper bound must be at least as ",
                      "big as the corresponding lower bound.")))
  }
  # set seed if none is provided by the user
  if ( is.null(seed) ) {
    seed <- as.numeric(Sys.time())
  }
  set.seed(seed)

  lst <- vector(mode = "list", length = length(vecs))
  for ( i in seq_along(vecs) ) {
    lst[[i]] <- runif(vecs[i], min = lower[i], max = upper[i])
  }
  lst
}
```

Use `Map()` to refactor the code for `randomSims2()` so as to avoid using the loop.

9. Supposing that `y` is a numerical vector, explain in words what the following code produces:

```
Filter(function(x) x >= 4, y)
```

10. Write a line of code using the sub-setting operator `[` produces the same result as the code in the previous problem.

11. Use `Filter()` to write a function called `oddMembers()` that, given any numerical vector, returns a vector containing the odd numbers of the given vector. Your function should take a single argument called `vec`, the given vector. A typical example of use would be as follows:

```
oddMembers(vec = 1:10)
```

```
## [1] 1 3 5 7 9
```

12. You are given the following list of character vectors:

```
   lst <- list(c("Akash", "Bipan", "Chandra", "Devadatta", "Raj"),
        c("Raj", "Vikram", "Sita", "Akash", "Chandra"),
        c("Akash", "Raj", "Chandra", "Bipan", "Lila"),
        c("Akash", "Vikram", "Devadata", "Raj", "Lila"))
```

Use `Reduce()` and the `union()` function to obtain a character vector that is the union of all the vectors in `lst`.

Chapter 14

Object-Oriented Programming in R

```
repeat {
    coffeeMug$drinkFrom()
    workTask$execute()
    if ( coffeeMug$empty() ) {
        if ( coffeePot$empty() ) {
            coffeePot$make()
        }
        coffeeMug$fill()
    }
    if ( workTask$done() ) {
        break
    }
}
```

Our journey with R began in the Procedural Programming paradigm. In the last Chapter we learned about R's extensive support for the more modern paradigm known as Functional Programming. In the present Chapter we will explore the ways in which R supports a third modern programming paradigm: Object-Oriented Programming.

14.1 The Object-Oriented Programming Paradigm

We started out with Procedural Programming, which draws a distinction between data and procedures. Data provides our information, and procedures solve problems by manipulating data into new forms.

Functional Programming puts functions at the center of attention. As much as possible, procedures are wrapped up in function calls, and by their status as first-class citizens (data) functions become king. Everything that happens is a result of function calls on data–where the data can include other functions.

You could say that Object-Oriented Programming reverses the point of view, attempting to put objects–conceived as complex forms of data–at the center of attention.

There are two major types of Object-Oriented (OO) Programming:

- **Message-Passing OO**
- **Generic-Function OO**

Message-Passing OO is the type of OO Programming that programmers usually associate with the term "object-oriented", and since it is the type that you will meet most frequently in other languages that support OO-programming we will discuss it first. Generic-Function OO is actually the older type of OO Programming. Although it is very important for understanding some aspects of how R works, it is met with less frequently in other major programming languages, so we will defer discussion of it until Section 14.7. Accordingly, the following introductory remarks pertain only to message-passing OO.

Objects in the world about us are indeed complex entities: they have particular features. Some of these features they have in virtue of the kind of object that they are, and others are features–*attributes*–that they have on their own, as individuals. And they can do various things. Again, some of the things they are do are things that they do simply because of the kind of thing they are, and others things they do might be unique to them.

Consider a typical object: my dog, for example. He has various attributes: his name is Beau, he has four legs, he weighs about 50 pounds, his hair is black. And he does various things—he has various *methods*, let's say, for making his way through life. For example, when we are out of the house he lies in the sofa, when we are home he lies on the floor. He barks. He eats. He annoys cats.

Some of his attributes and methods Beau has simply in virtue of being the kind of thing that he is. He is, after all, a dog—a member of the class Dog, let's say—and so like all other dogs he will have four legs and he will be able to bark and to annoy cats. Of course all dogs are animals—members of the more general class Animal, let's say—and all animals eat, so as an animal Beau *inherits* the method known as eating. Of course all animals are material beings—members of the even-more general class Being, let's say—and all beings have a weight, and thus Beau has a weight, but the particular value of that weight—50 pounds—is determined by the way he has lived his life thus far. Other dogs, animals and material beings a weight, but they won't necessarily have the same weight as Beau.

What emerges from this discussion is an (admittedly oversimplified) view of the world as a collection of complex objects. These objects are related to one another in systematic ways–related closely when they are, for instance, both Dogs, and more more distantly, as when one is a Dog and another is a Cat. And these objects can act on each themselves and each other in various ways, through their methods.

The idea of the Object-Oriented Programming is that since we are so used to viewing the *world* as a collection of objects that are constituted of various attributes and methods and that are related to one another through membership in classes of varying levels of generality, then it might be help us to be able to view a *computer program* similarly—as a collection of objects thus constituted and thus related to one another.

Like many other modern programming languages, R provides considerable support for the Object-Oriented paradigm. In fact it provides that support in two different formats:

- Reference Classes
- R6 Classes

Reference Classes are a relatively recent introduction to R. R6 Classes are a simple and lightweight version of Reference Classes. They are not a part of the core of R—they are enabled by a contributed R-package—but they are quite useful as an introduction to object-oriented programming. Accordingly, we will work with them in this Chapter.

14.2 Reference Classes with Package R6

The package **R6** (Chang, 2017) provides access to R6 classes. You will need to attach it:

```
library(R6)
```

We'll get started by creating a miniature computerized world of objects that is based on characters from the Wizard of Oz.

14.2.1 Defining a Class

Construction of the object-world begins with the definition of classes. A *class* is simply a general prototype on the basis of which actual objects may be created.

Let's define the class `Person`. This is accomplished with the `R6Class()` function:

```
Person <- R6Class(classname = "Person",
                  public = list(
                    name = NULL,
                    age = NULL,
                    desire = NULL,
                    initialize = function(name = NA, age = NA, desire = NA) {
                      self$name <- name
                      self$age <- age
                      self$desire <- desire
                      self$greet()
                    },
                    set_age = function(val) {
                      self$age <- val
                      cat("Age of ", self$name, " is:  ", val, ".\n", sep = "")
                    },
                    set_desire = function(val) {
                      self$desire <- val
                      cat("Desire of ", self$name, " is:  ", val, ".\n", sep = "")
                    },
                    greet = function() {
                      cat(paste0("Hello, my name is ", self$name, ".\n"))
                    }
                  )
)
```

Let's analyze the above code. The function `R6Class()` has a quite a few parameters, but the two with which we will concern ourselves the most are:

- `classname`: the name we propose to give to the class;

- `public`: a list of members of the class that are "public" in the sense that they can be accessed and used outside of the class.[1] Members are of two types:
  - *attributes*: the objects `name`, `age` and `desire`. Currently these have value `NULL`, but they can be given other values later on. What makes then attributes, though, is that they will not be given a function as a value.
  - *methods*: these are members of the class that are functions. You can think of a "method" as a particular way of performing a common task that could be performed in many different ways. The four methods you see in `Person` are:
  - `initialize`: This is a function that will be run whenever a new object of class Person is created. As we can see from its arguments, it will be possible to give values to the attributes `name`. `age` and `desire` when this function is called.
  - `set_age` and `set_desire` are functions that allow a user to set or to change the value of `age` and `desire` for an object of class Person.
  - `greet`: a function that will permit an object of class Person to issue a greeting.

Note the use of the term `self` in the code for the methods of the class. When a method is called on an object, the term `self` will refer to the object on which the method is being called. Hence, for example, in the code for `greet` the term `self$name` will evaluate to the name of the person who issues the greeting.

14.2.2 Instantiation: Initializing Objects

On its own there is not much that a class can *do*. For anything to happen we need to create a particular individual: an object of class Person. Creation of an object having a particular class is called *instantiation*.

Every class comes with a method—not mentioned in the definition of the class—called `new()`. This method, when called on the Class, instantiates an object of the class by running the `initialize()` function. Let's create a person named Dorothy and store this new object in the variable `dorothy`:

```
dorothy <- Person$new(name = "Dorothy", age = 12,
                      desire = "Kansas")
```

```
## Hello, my name is Dorothy.
```

Note how the dollar-sign is used to indicate the calling of the `new()` method on the class Person. Note also that the arguments of `new()` are the arguments of `initialize()`: that's because the `new()` method actually runs `initialize()` as part of its object-creation process.

We can get a look at `dorothy` by printing her to the console:

```
dorothy
```

```
## <Person>
##   Public:
##     age: 12
##     clone: function (deep = FALSE)
##     desire: Kansas
##     greet: function ()
##     initialize: function (name = NA, age = NA, desire = NA)
##     name: Dorothy
```

[1] `R6Class()` has another parameter called `private`, which takes as value a list of members that can only be accessed within the class by members of the class, not by programmers working outside of the class itself. In the course of development of very large and complex programs, it can be useful to keep some members private so that programmers don't accidentally change too much about the way objects work. Since our programs are still on the small side, we won't worry about private members, for now.

```
##     set_age: function (val)
##     set_desire: function (val)
```

We get the basic information on Dorothy, including also an indication that she can be cloned (copied). We'll discuss cloning later.

We can instantiate as many people as we like. The code below for example, establishes `scarecrow` as a new instance of Person:

```
scarecrow <- Person$new(name = "Scarecrow", age = 0.038,
                        desire = "Brains")
```

```
## Hello, my name is Scarecrow.
```

In *The Wizard of Oz*, Scarecrow is only two weeks old when Dorothy meets him, so his age is set at $2/52 = 0.38$ years.

14.2.3 Getting and Setting Attributes

If we would like to change Dorothy's age, we can do so by calling the `set_age()` method on her:

```
dorothy$set_age(13)
```

```
## Age of Dorothy is:  13.
```

```
dorothy$age
```

```
## [1] 13
```

We can also set Dorothy's age directly, by regular assignment:

```
dorothy$age <- 14
```

```
dorothy$age
```

```
## [1] 14
```

The effect is the same as when we use `set_age()` except that we don't get a report to the console.

14.2.4 Calling Methods

We have seen that in order to call a method on an object, you follow the format:

```
object$method()
```

Thus, we can ask `dorothy` to issue a greeting:

```
dorothy$greet()
```

```
## Hello, my name is Dorothy.
```

In the syntax for calling methods, we see one aspect of "message-passing": `dorothy$greet()` essentially passes a message to `dorothy`: "Please call your `greet()` method!"

14.2.5 Holding a Reference, not a Value

R6 objects operate by what are known in computer programming as *reference semantics*. This means that when the assignment operator is used to assign an R6 object to a variable, the new variable holds a *reference* to that object, not a copy of the new object. This is in contrast *value semantics*, which is the way assignment usually works in R. In value semantics, R a distinct copy of the value of the assigned object is created, and the new variable refers to that copy.

Below is an example of the familiar value semantics in action:

```
a <- 10
b <- a
b <- 20
a
```

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

`b` has been changed to 20, but that change did not affect `a`, which keeps its initial value of 10.

We can use the function `address()` from the package **pryr** (Wickham, 2015) to track what is happening behind the scenes. `address()` will tell us the current location in memory of the value corresponding to a name. Let's repeat the above process, but use `address()` to see where the values are stored:

```
a <- 10
pryr::address(a)
```

```
## [1] "0x10ce0be88"
```

The value 10 is stored in the memory address given above. Next, let's create `b` by assignment from `a`:

```
b <- a
pryr::address(b)
```

```
## [1] "0x10ce0be88"
```

For the moment `b` points to the same place in memory as `a` does, so it will yield 10:

```
b
```

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

We can use `b` for other operations; for example, we can add 30 to it:

```
b + 30
```

```
## [1] 40
```

But for now `b` still points to the same place in memory that `a` does:

```
pryr::address(b)
```

```
## [1] "0x10ce0be88"
```

But now let's assign a new value to `b`:

```
b <- 20
pryr::address(b)
```

```
## [1] "0x12b9889c8"
```

Aha! `b` now points to a **new** location in the computer's memory! That's because R saw that the assignment operator was going to change the value of `b`. Since `b` has value semantics, R knows to set aside a new spot in memory to contain the value 20, and to associate the name `b` with that spot. That way, the new `b` won't interfere with `a`, which points to the same old spot in memory and thus remains 10:

```
pryr::address(a)
```

```
## [1] "0x10ce0be88"
```

```
a
```

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

On the other hand, R6 objects have *reference semantics*. We can see this in action with the following example. First, let's check on `dorothy`'s age:

```
dorothy$age
```

```
## [1] 14
```

Let's now create `dorothy2` by assignment from `dorothy`:

```
dorothy2 <- dorothy
```

Let's check the memory locations:

```
c(pryr::address(dorothy), pryr::address(dorothy2))
```

```
## [1] "0x1297401b8" "0x1297401b8"
```

As with `a` and `b`, the two names initially point to the same place in memory.

Now let's change the age of `dorothy2` to 30:

```
dorothy2$age <- 30
```

Let's check the age of `dorothy`:

```
dorothy$age
```

```
## [1] 30
```

Whoa! Changing the age of `dorothy2` changed the age of `dorothy`! That's because of the reference semantics: `dorothy2` continues to be associated with the same spot in memory as `dorothy`, *even after* we begin to make changes to it:

```
c(pryr::address(dorothy), pryr::address(dorothy2))
```

```
## [1] "0x1297401b8" "0x1297401b8"
```

14.2.6 Cloning an Object

For the sorts of complex objects created by R6 classes, reference semantics can be a useful feature. But what if we want a new and truly distinct copy of an R6 object? For this we need the `clone()` method that was alluded to earlier:

```
dorothy2 <- dorothy$clone()
dorothy2
```

```
## <Person>
##   Public:
##     age: 30
##     clone: function (deep = FALSE)
##     desire: Kansas
##     greet: function ()
##     initialize: function (name = NA, age = NA, desire = NA)
##     name: Dorothy
##     set_age: function (val)
##     set_desire: function (val)
```

`dorothy2` looks just like `dorothy`. However, the name `dorothy2` does not point to the same place in memory:

```
c(pryr::address(dorothy), pryr::address(dorothy2))
```

```
## [1] "0x1297401b8" "0x12a77eb90"
```

Accordingly, changes to `dorothy2` will no longer result in changes to `dorothy`:

```
dorothy2$age <- 100
dorothy$age
```

```
## [1] 30
```

You should know that if one or more of the members of an object is itself an object with reference semantics (such as an instance of an R6 class), then a copy produced by `clone()` will hold a reference to that same member-object, not a separate copy of the member-object. In such a case if you still want a totally separate copy, you will have to consult the R6 manual on the topic of "deep cloning."

14.3 Inheritance

Earlier we mentioned that in the real world an object may possess a property in virtue of being the particular kind of thing that it is, or it may possess that property in virtue of being a member of some more general class of things. Thus a dog can eat, not because it is a dog *per se*, but because it is an animal. You could say that dogs "inherit" the capacity to eat from the class Animal.

Most object-oriented frameworks allow for some type of inheritance between classes, and R6 is no exception. In the *Wizard of Oz* lions appear to be a special type of person: after all, they can speak, and they have desires. Accordingly, we should create a new class Lion that inherits from the class Person:

```
Lion <- R6Class("Lion",
                inherit = Person,
                public = list(
                  weight = NULL,
                  set_weight = function(val) {
                    self$weight <- val
                    cat(self$name, " has weight:  ", val, ".\n", sep = "")
                  },
                  greet = function() {
                    cat("Grr!  My name is ", self$name, "!", sep = "")
                    }
                  )
                )
```

The key to inheritance is the new argument in the code above:

```
inherit = Person
```

The impact of this additional argument is that any instance of the class Lion will possess by default all of the attributes and methods of class Person, in addition to the new attributes and methods (`weight`, `set_weight`, and `greet`) specified explicitly in the definition of class Lion. Of these three explicitly-defined members:

- `weight` and `set_weight`were not already in Person, so they are simply added on as members of Lion;[2]
- `greet` was already a member of Person, but the new specification of `greet` in the definition of Lion *overrides* the old one. Any instance of class Lion will greet us according to the code for `greet()` given in the definition of Lion. (Objects that are instances of the class Person, however, will go on greeting us just as they always have done.)

Let's verify all of this by making an instance of class Lion:

```
cowardlyLion <- Lion$new(name = "Cowardly Lion",
                         age = 18, desire = "courage")
```

```
## Grr!  My name is Cowardly Lion!
```

Since the definition of class Lion did not specify a new method of initialization—we could have asked for this, but chose not to—the instantiation of a lion won't set its weight. We will have to do that separately:

```
cowardlyLion$set_weight(350)
```

```
## Cowardly Lion has weight:  350.
```

Inheritance is an very effective way to re-use code and to keep code for a particular function in one place. Once we have written methods for some class X, those methods will be available to any objects that are instances of classes that inherit from class X. Hence there is no need to repeat the code in the definition of the inheriting classes. Later on if we need to modify one of of the class X methods, we need only make the change in class X: we don't have to worry about changing it in all of the classes that inherit from X.

Let's ask `cowardlyLion` to greet us:

[2] In the real world any person—not just any lion—has a weight. By making weight a feature specific to class Lion we are simply indicating that in our program weight is of particular importance to lions, but not to people in general.

```
cowardlyLion$greet()
```

```
## Grr!  My name is Cowardly Lion!
```

As expected, the greeting includes a growl, since the `greet()` method for class Lion overrides the growl-less `greet()` method that it would have otherwise inherited from class Person. We also see a fundamental aspect of message-passing OO at work:

> The method used for determining how a particular task is performed is determined by the class of the object that is asked to perform the task.

Here we have the task of greeting, which is common to objects of class Person and to objects of class Lion. That task may be performed with or without a growl. Precisely how it is performed—the method that is selected for carrying out the task—depends upon the class of the object that performs the greeting: an object of class Person will greet you without a growl, whereas an object of class Lion will greet you with a growl.

The common task of greeting may be performed in different ways, depending upon the class of the object involved in the task. This is an example of what programmers call *polymorphism*. The term "polymorphism" comes from a Greek word that means "many forms", and it refers to a task being performed differently depending on the class of the object involved. Polymorphism makes programming a bit easier: you only have to remember the name of the one common task. Invoke it on an object and you can rest assured that the task will be performed in the way that is right and proper to that object.

14.4 Adding Members to a Class

Sometimes we need to add or to modify a member of a class, after we have defined that class. Every class comes with a `set()` method that allows us to do this. For example, suppose that we would like our people to have a favorite color:

```
Person$set("public", "color", NA, overwrite = TRUE)
```

We now see that `color` is now one of the public members of class Person:

```
Person
```

```
## <Person> object generator
##   Public:
##     name: NULL
##     age: NULL
##     desire: NULL
##     color: NA
##     initialize: function (name = NA, age = NA, desire = NA)
##     set_age: function (val)
##     set_desire: function (val)
##     greet: function ()
##     clone: function (deep = FALSE)
##   Parent env: <environment: R_GlobalEnv>
##   Locked objects: TRUE
##   Locked class: FALSE
##   Portable: TRUE
```

In the above call to set, the argument `overwrite = TRUE` was not strictly necessary, since `color` did not previously exist as a member of Person. If you are developing a program, though, and you find yourself

repeatedly running a piece of code that sets a new member for a class, it's useful to have overwriting turned on.

While we are at it. let's write a special `set_color()` method:

```
Person$set("public", "set_color", function(val) {
  self$color <- val
  cat(self$name, " has favorite color: ", val, ".\n", sep = "")
  }, overwrite = TRUE)
```

You might think that we can now set a favorite color for `dorothy`:

```
dorothy$set_color("blue")
```

```
Error: attempt to apply non-function
```

Why did this not work? It turns out that when you add a new member to a class it is only available to instances of the class that are created **after** the new member is added to that class. If we create a new instance of Person, the Good Witch Glinda, let's say, then we should be able to give her a favorite color:

```
glinda <- Person$new("Glinda", "500", "the good of all")
```

```
## Hello, my name is Glinda.
```

```
glinda$set_color("blue")
```

```
## Glinda has favorite color: blue.
```

14.5 Method Chaining

In this Section we will explore a concise and convenient device for calling multiple methods on an object. The device is known as *method chaining*.

In order to illustrate method chaining, we'll extend the class Lion a bit, with some new attributes and new methods. First of all, we'd like to add a data frame consisting of possible animals on which a lion might prey. The data frame will contain the name of each type of animal, and an associated weight.

```
animal <-  c("zebra", "giraffe", "pig",
             "cape buffalo", "antelope", "wildebeast")
mass <- c(50, 100, 25, 60, 45, 55)
prey <- data.frame(animal, mass, stringsAsFactors =FALSE)
```

Let's now add `prey` as a new attribute to class Lion. We will also add an attribute `eaten`: a character vector—initially empty—that will contain a record of all the animals that the lion has eaten.

```
Lion$set("public", "prey", prey, overwrite = TRUE)
Lion$set("public", "eaten", character(), overwrite = TRUE)
```

Let us now endow our lions with the capacity to eat a beast of prey, by adding the method `eat()`:

```
Lion$set("public", "eat", function() {
  n <- nrow(self$prey)
  item <- self$prey[sample(1:n, size = 1), ]
  initLetter <- substr(item$animal, 1, 1)
  article <- ifelse(initLetter %in% c("a", "e", "i", "o", "u"), "An ", "A ")
  cat(article, item$animal, " was eaten just now ...\n\n", sep = "")
  self$eaten <- c(self$eaten, item$animal)
  self$weight <- self$weight + item$mass
  return(invisible(self))
}, overwrite = TRUE)
```

When it eats an animal—randomly selected from the `prey` data frame—the lion gains the amount of weight that is associated with its unfortunate victim, and the victim is added to the lion's `eaten` attribute.

Note that the `eat()` method returns `self` invisibly. When the method is called on a lion, that very lion itself is returned as a value, but since it is returned invisibly it won't be printed to the console. The usefulness of returning `self` will soon become apparent.

Lions enjoy talking about what they have eaten recently, and for some reason they monitor their weight obsessively. The new method `report()` accounts for these characteristics of lions:

```
Lion$set("public", "report", function() {
  n <- length(self$eaten)
  if ( n >= 1 ) {
    cat("My name is ", self$name, ".\n", sep = "")
    cat("My most recent meal consisted of: ", self$eaten[n], ".\n", sep = "")
  }
  cat("I now weigh ", self$weight, " pounds.\n", sep = "")
  return(invisible(self))
}, overwrite = TRUE)
```

Note that `report()` also returns the lion invisibly.

Let us now instantiate a new lion named Simba:

```
simba <- Lion$new(name = "Simba", age = 10,
                  desire = "Hakuna Matata")
```

```
## Grr!  My name is Simba!
```

```
simba$set_weight(300)
```

```
## Simba has weight:  300.
```

Having been created after the addition of the `prey` and `eaten` attributes and the `eat()` and `report()` methods, `simba` has access to all of them. in particular, he can eat a random beast of prey:

```
simba$eat()
```

```
## A cape buffalo was eaten just now ...
```

`simba` has eaten, and he has presumably gained some weight as a result. Let's see verify this by asking for his report:

```
simba$report()
```

```
## My name is Simba.
## My most recent meal consisted of: cape buffalo.
## I now weigh 360 pounds.
```

Let's now have `simba` eat twice more, and then report. Because `eat()` and `report()` both return `simba`, we can

- call `eat()` on 'simba,
- and then immediately call `eat()` on the result,
- and finally call `report()` on the result of our second call.

All of this can be accomplished in one line of calls, in which the three method-calls are "chained" together with dollar-signs:

```
simba$eat()$eat()$report()
```

```
## A pig was eaten just now ...
##
## A cape buffalo was eaten just now ...
##
## My name is Simba.
## My most recent meal consisted of: cape buffalo.
## I now weigh 445 pounds.
```

In object-oriented programming languages you will see method-chaining used quite frequently.

14.6 Application: Whales in an Ocean

Let's now write a more elaborate program in the object-oriented style. Our program will simulate the population growth for whales in a ocean. In our model:

- Whales are mostly solitary: they move randomly about the ocean, happily feeding upon plankton.
- When a male and female whale are sufficiently close together the female whale will check out the male to see if he is mature enough to mate. If she herself is fertile, then she will mate with him and a child will be produced.
- That child will be either male or female, with equal likelihood for either option.
- After mating, the female will be infertile for a period of time.
- Whales have a set maximum lifetime.
- in any given time period, it is possible for a whale to starve, causing it to die before reaches its lifespan. The probability of starvation is low when the whale population is small (presumably there is plenty of plankton available then) but the starvation-probability increases in direct proportion to the population.

From the above conditions you can see that if the whale population is small then there is low probability of starvation, but on the other hand whales are liable to be spread thinly throughout the ocean. As a result males and females are relatively unlikely to run across each other, and hence births are less likely to occur. With a low enough population there is a strong possibility that the whales will die off before they can reproduce. (This is why biologists worry about some whale populations: if the whales are hunted to the point where the population is below a certain critical threshold, then they can go extinct on their own, even if hunting ceases.) On the other hand, if the whale population grows large then males and female meet frequently, and there are plenty of births. On the other hand there is a general shortage of food, resulting in high starvation rates. At some point birth and starvation balance each other out, resulting in a long-term equilibrium population-level.

We will implement our simulation with various objects:

- an ocean
- whales of two types:
  - male whales
  - female whales

We will begin by defining the class Whale:

```
Whale <- R6Class("Whale",
                 public = list(
                   position = NULL,
                   age = NULL,
                   lifespan = NULL,
                   range = NULL,
                   maturity = NULL,
                   stepSize = NULL,
                   initialize = function(position = NA, age = 3,
                                         lifespan = 40, range = 5,
                                         maturity = 10, stepSize = 5) {
                     self$position <- position
                     self$age <- age
                     self$lifespan <- lifespan
                     self$range <- range
                     self$maturity <- maturity
                     self$stepSize <- stepSize
                   }
                 ))
```

For whales that are instances of the Whale class:

- `position` will contain the current x and y-coordinate of the whale in a two-dimensional ocean;
- `age` will give the current age of the whale.
- `lifespan` is the maximum age that the whale can attain before it dies.
- `range` is how close the whale has to be to another whale of the opposite sex in order to detect the presence of that whale.
- `maturity` is the age that the whale must attain in order to be eligible to mate.
- `stepSize` is the distance that the whale moves in the ocean in a single time-period.

Whales need to be able to move about in the sea. A whale moves by picking a random direction—any angle between 0 and 2π radians—and then moving `stepSize` units in that direction. If the selected motion would take the whale outside the boundaries of the ocean, then R will repeat the motion until the whale lands properly within the boundaries.

The motion of a whale is implemented in the code below for a `move()` method that is added to class Whale:

```
Whale$set("public",
          "move",
          function(dims, r = self$stepSize) {
            xMax <- dims[1]
            yMax <- dims[2]
            repeat {
              theta <- runif(1, min = 0, max = 2*pi)
              p <- self$position + r * c(cos(theta), sin(theta))
              within <- (p[1] > 0 && p[1] < xMax) && (p[2] > 0 && p[2] < yMax)
```

```
            if ( within ) {
              self$position <- p
              break
            }
          }
        }, overwrite = TRUE)
```

Note the parameter `dims` for the move function. From the code we can tell that it's a vector of length 2. In fact the parameter will contain the x and and y-dimensions (the width and breadth) of our ocean. It's something that will have to be determined by the ocean itself. We haven't written the class Ocean yet, so this part of the code will have to remain a bit mysterious, for now.

We need male and female whales, so we create a class for each sex. Both classes inherit from Whale. The class Male adds only a `sex` attribute:

```
Male <- R6Class("Male",
                inherit = Whale,
                public = list(
                  sex = "male"
                ))
```

A female whale is a bit more complex: in addition to a sex attribute, she needs an attribute that specifies how long she will be infertile after giving birth, and another attribute that enables the program to keep track of the number of time-periods she must wait until she is fertile again:

```
Female <- R6Class("Female",
                inherit = Whale,
                public = list(
                  sex = "female",
                  timeToFertility = 0,
                  infertilityPeriod = 5
                ))
```

A female whale also needs a method to determine whether a mature whale is in the vicinity:

```
Female$set("public",
           "maleNear",
           function(males, dist) {
             foundOne <- FALSE
             for ( male in males ) {
               near <- dist(male$position, self$position) < self$range
               mature <- (male$age >= male$maturity)
               if ( near && mature ) {
                 foundOne <- TRUE
                 break
               }
             }
             foundOne
           }, overwrite = TRUE)
```

Again, note the parameters `males` and `dist`. Values for these parameters will be provided by the ocean object. `males` will be a list of the male whales in the population at the current time, and `dist` will be a

function for computing the distance between any two points in the ocean.

A female whale also needs to be able to mate:

```
Female$set("public",
           "mate",
           function() {
             babySex <- sample(c("female", "male"), size = 1)
             self$timeToFertility <- self$infertilityPeriod
             return(babySex)
           }, overwrite = TRUE)
```

Now it is time to define the Ocean class. It's a bit of a mouthful:

```
Ocean <- R6Class("Ocean",
                 public = list(
                   dimensions = NULL,
                   males = NULL,
                   females = NULL,
                   malePop = NULL,
                   femalePop = NULL,
                   starveParameter = NULL,
                   distance = function(a, b) {
                     sqrt((a[1] - b[1])^2 + (a[2] - b[2])^2)
                   },
                   initialize = function(dims = c(100, 100),
                                         males = 10,
                                         females = 10,
                                         starve = 5) {
                     self$dimensions <- dims
                     xMax <- dims[1]
                     yMax <- dims[2]
                     maleWhales <- replicate(
                       males,
                       Male$new(age = 10,
                                position = c(runif(1, 0, xMax),
                                             runif(1, 0, yMax))))
                     femaleWhales <- replicate(
                       females,
                       Female$new(age = 10,
                                  position = c(runif(1, 0, xMax),
                                               runif(1, 0, yMax))))
                     self$males <- maleWhales
                     self$females <- femaleWhales
                     self$malePop <- males
                     self$femalePop <- females
                     self$starveParameter <- starve
                   },
                   starvationProbability = function(popDensity) {
                     self$starveParameter * popDensity
                   }
                 ))
```

In an instantiation of the class Ocean:

- `dimensions` will be numerical vector of length 2 that specifies the width and breadth of the ocean.
- `males` and `females` will be lists that contain respectively the current sets of male whales and female whales. Note that this implies that an ocean will contain as members other items that are themselves R6 classes. This is an example of what is called *composition* in object-oriented programming.
- `malePop` and `femalePop` give respectively the current counts male and females whales.
- `starveParameter` helps determine the probability that each individual whale will starve within the next time-period. Note that the method `starvationProbability()` makes the probability of starvation equal product of `starveParameter` and the current density of the population of whales in the ocean. The bigger this attribute is, the more starvation will occur and the lower the long-term upper limit of the population will be.
- `distance()` is the method for finding the distance between any two positions in the ocean. It implements the standard distance-formula from high-school geometry.
- The initialization function permits the user to determine the dimensions of the ocean, the initial number of male and female whales, and the starvation parameter. In the instantiation process for an individual ocean the required number of male and female whales are instantiated and are placed randomly in the ocean.

We need to add a method that allows an ocean to advance one unit of time. During that time:

- Each mature and fertile female whale must check for nearby mature males, mate with one if possible, and produce a baby.
- Any offspring produced must then be added to the ocean's lists of male and female whales.
- The ocean must then subject each whale to the possibility of starvation within the time-period at hand.
- All whales that survive must then move.
- The age of each whale must be increased by one time-unit. For females, the time remaining until fertility must also be decreased by a unit.

The `advance()` method is implemented and added to Ocean with the code below:

```
Ocean$set("public",
          "advance",
          function() {
            malePop <- self$malePop
            femalePop <- self$femalePop
            population <- malePop + femalePop
            if ( population == 0 ) {
              return(NULL)
            }
            males <- self$males
            females <- self$females
            babyMales <- list()
            babyFemales <- list()
            if ( malePop > 0 && femalePop > 0 ) {
              for ( female in females ) {
                if ( female$age >= female$maturity &&
                     female$timeToFertility <= 0 &&
                     female$maleNear(males = males,
                                     dist = self$distance)) {
                  outcome <- female$mate()
                  if ( outcome == "male" ) {
                    baby <- Male$new(age = 0, position = female$position)
                    babyMales <- c(babyMales, baby)
                  } else {
                    baby <- Female$new(age = 0, position = female$position)
                    babyFemales <- c(babyFemales, baby)
```

```
      }
    }
  }
}

# augment the male and female lists if needed:
lmb <- length(babyMales); lfb <- length(babyFemales);

# throw in the babies:
if ( lmb > 0 ) {
  males <- c(males, babyMales)
}
if ( lfb > 0 ) {
  females <- c(females, babyFemales)
}

# revise population for new births:
population <- length(males) + length(females)

# starve some of them, maybe:
popDen <- population/prod(self$dimensions)
starveProb <- self$starvationProbability(popDensity = popDen)
maleDead <- logical(length(males))
femaleDead <- logical(length(females))
# starve some males
for ( i in seq_along(maleDead) ) {
  male <- males[[i]]
  maleDead[i] <- (runif(1) <= starveProb)
  male$age <- male$age + 1
  if ( male$age >= male$lifespan ) maleDead[i] <- TRUE
  if ( maleDead[i] ) next
  # if whale is not dead, he should move:
  male$move(dims = self$dimensions)
}
# starve some females
for ( i in seq_along(femaleDead) ) {
  female <- females[[i]]
  femaleDead[i] <- (runif(1) <= starveProb)
  female$age <- female$age + 1
  if ( female$age >= female$lifespan ) femaleDead[i] <- TRUE
  if ( femaleDead[i] ) next
  if ( female$sex == "female" ) {
    female$timeToFertility <- female$timeToFertility - 1
  }
  # if female is not dead, she should move:
  female$move(dims = self$dimensions)
}

# revise male and female whale lists:
malePop <- sum(!maleDead)
self$malePop <- malePop
femalePop <- sum(!femaleDead)
```

```
            self$femalePop <- femalePop
            if ( malePop > 0 ) {
              self$males <- males[!maleDead]
            } else {
              self$males <- list()
            }
            if ( femalePop > 0 ) {
              self$females <- females[!femaleDead]
            } else {
              self$females <- list()
            }
          }, overwrite = TRUE)
```

In simulations we might enjoy looking at a picture of the ocean at any given time moment. Hence we add a `plot()` method that enables an ocean to produce a graph of the whales within it. In this graph males will be colored red and females green. Mature whales will appear as larger than immature whales. For purposes of speed we use R's base graphics system rather than **ggplot2**.

```
Ocean$set("public",
          "plot",
          function() {
            males <- self$males[1:self$malePop]
            females <- self$females[1:self$femalePop]
            whales <- c(males, females)
            if ( length(whales) == 0) {
              plot(0,0, type = "n", main = "All Gone!")
              box(lwd = 2)
              return(NULL)
            }
            mat <- sapply(whales, function(x) {
              c(x$position[1], x$position[2],
                as.numeric(x$sex == "male"), as.numeric(x$age >= x$maturity))
              }, simplify = "array")
            mat <- simplify2array(t(mat))
            df <- as.data.frame(mat)
            names(df) <- c("x", "y", "sex", "mature")
            # males will be red, females green:
            df$color <- ifelse(df$sex == 1, "red", "green")
            # mature whales have cex = 3, immature whales cex 0.7
            df$size <- ifelse(df$mature == 1, 1.3, 0.7)
            with(df,
                 plot(x, y, xlim = c(0, self$dimensions[1]),
                 ylim = c(0, self$dimensions[1]), pch = 19, xlab = "",
                 ylab = "", axes = FALSE, col = color, cex = size,
                 main = paste0("Population = ", nrow(df))))
            box(lwd = 2)
          }, overwrite = TRUE)
```

Finally, we write a simulation function. The function allows the user to specify the number of time-units that the simulation will cover, along with initial numbers of male and female whales. The user will have an option to "animate" the simulation showing a plot of the ocean after each time-unit. If the animation option is chosen, then R will use the `Sys.sleep()` function to make the computer suspend computations for half

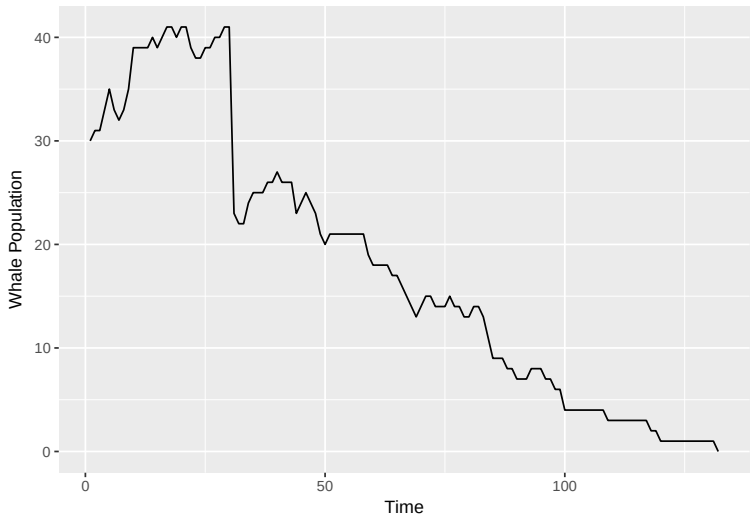


Figure 14.1: Simulated whale population. Sadly, they all die after about 130 days.

a second so that the user can view the plot. The simulation will cease if all of the whales die prior to end of the allotted time. Finally, the function uses **ggplot2** to produce a graph of the whale population as a function of time.

```
library(ggplot2)
oceanSim <- function(steps = 100, males = 10, females = 10,
                     starve = 5, animate = TRUE, seed = NULL) {
  if ( !is.null(seed) ) {
    set.seed(seed)
  }
  ocean <- Ocean$new(dims = c(100, 100), males = males,
                     females = females, starve = starve)
  population <-numeric(steps)
  for ( i in 1:steps ) {
    population[i] <- ocean$malePop + ocean$femalePop
    if ( animate ) ocean$plot()
    if ( population[i] == 0 ) break
    ocean$advance()
    if ( animate ) Sys.sleep(0.5)
  }
  pop <- population[1:i]
  df <- data.frame(time = 1:length(pop),
                   pop)
  ggplot(df, aes(x = time, y = pop)) + geom_line() +
    labs(x = "Time", y = "Whale Population")
}
```

We are now ready for a simulation. The results are graphed in Figure 14.1.

```
oceanSim(steps = 200, males = 15, females = 15,
         animate = FALSE, seed = 3030)
```

You should try running the simulation yourself a few times, for differing initial numbers of whales. If you like you can watch the whales move about by setting `animate` to `TRUE`:

```
oceanSim(steps = 200, males = 15, females = 15,
         animate = TRUE)
```

You might also wish to explore varying the `starve` parameter: recall that the higher it is, the lower the long-term stable whale population will be. In order to better detect long-term stable population sizes, you will want to work with more steps, and you should turn off the step-by-step-animation, thus:

```
oceanSim(steps = 1000, males = 60, females = 60,
         starve = 2.5, animate = TRUE)
```

The object-oriented approach to simulation is not necessarily the quickest approach: R6 objects do require a bit more time for computation, as compared to a system that stores relevant information about the population in vectors or in a data frame. On the other hand the object-oriented approach makes it easy to encode information about each individual whale as it proceeds through time. In larger applications, an object-oriented approach to programming can result in code that is relatively easy to read and to modify, albeit at some cost in terms of speed.

14.7 Generic-Function OO

We now turn to the second major type of object-oriented programming that is supported by R, namely: generic-function OO.

14.7.1 Motivating Examples

We begin by revisiting the task of *printing* to the console.

Recall that whenever we type the name of an object into the console and press Enter, R interprets the input as a call to the `print()` function. Consider, for example, printing out some portion of `m111survey` from the **tigerstats** package.

Thus the following two expressions accomplish the same

```
df <- tigerstats::m111survey[, c("height", "weight_feel")]
```

If we want to print `df`, either of the following two statements will suffice:

```
print(df)
df          # calls print(df) implicitly
```

Let's have a look at `df`, using either of the above two commands:

```
##   height   weight_feel
## 1   76.00 1_underweight
## 2   74.00 2_about_right
## 3   64.00 2_about_right
## 4   62.00 1_underweight
## 5   72.00 1_underweight
<further output omitted to save space>
```

It is tempting to think of the above printout as simply what the object `df` *is*. But that's not quite right. In truth, it merely reflects how R chose to *represent* `df` to us in the console. R was programmed to represent `df`

in spreadsheet-format—with variables along columns, individuals along rows, and with handy row-numbers supplied—because users are accustomed to viewing data frames in that way.

Now let's turn `df` into a list:

```
lst <- as.list(df)
str(lst)
```

```
## List of 2
##  $ height     : num [1:71] 76 74 64 62 72 70.8 70 79 59 67 ...
##  $ weight_feel: Factor w/ 3 levels "1_underweight",..: 1 2 2 1 1 3 2 2 2 3 ...
```

Now let's print `lst`:

```
lst   # same as print(lst)
```

```
## $height
##  [1] 76.00 74.00 64.00 62.00 72.00 70.80 70.00 79.00 59.00 67.00 65.00 62.00
## [13] 59.00 78.00 69.00 68.00 73.00 73.00 65.00 65.00 66.00 67.75 63.00 66.00
<further output omitted to save space>
```

We get the familiar output for a list whose elements are named. Users don't expect lists to be represented in the console in spreadsheet-style format, even if the elements of the list happen to be vectors that are all of the same length. They expect a more "neutral" representation, and R delivers one.

Printing to the console is a common task. It appears, however, that the method by which that task is performed depends on the type of object that is input to the `print()` function. If your object is a data frame, `print()` behaves one way. If your object is a list, print does something else. Since the behavior of `print()` depends on the type of object involved in the operation of printing, you could say that it exhibits polymorphism. .

In fact it is the *class* of the object given to `print()` that determines the method that `prints()` employs. The class of an object in R can be accessed with the `class()` function :

```
class(df)
```

```
## [1] "data.frame"
```

```
class(lst)
```

```
## [1] "list"
```

How does the class of `df` determine the method used for printing? To see how this is done, look at the code for the `print()` function:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x108da3d40>
## <environment: namespace:base>
```

The body of the `print()` consists of just one expression: `UseMethod("print")`. On the fact of it, this doesn't seem to accomplish anything! In reality, though, a lot is taking place under the hood. Let's examine what happens, step-by-step, when we call `print(df)`.

1. The data frame `df` is assigned to the parameter `x` in the `print()` function.
2. We call `UseMethod("print")`.
3. From `help(UseMethod)` we learn that `UseMethod()` takes two parameters:
   - `generic`: a character string that names task we want to perform. In this case `generic` has been set to "print".
   - `object`: this is an object whose class will determine the method that will be "dispatched", i.e., the method that will be used to print the object to the console. By default this is the first argument in the enclosing function `print()`, so `object` gets set to the data frame `df`.
4. Control has now passed to the `UseMethod()` function, which searches for a suitable method for printing an object of class `data.frame`. It does this by pasting together "print" (the argument to `generic`) and `data.frame` (the class of the object `df` it was given) with a period in between, getting the string "print.data.frame". A search is now conducted for a function of the name.
5. The function `print.data.frame()` will be found. We can tell because it appears on the list of available "methods" for `print()`. The `methods()` function will give us the complete list of available methods, if we like

```
methods("print")
```

```
## [1] print,ANY-method
## [2] print,diagonalMatrix-method
## [3] print,sparseMatrix-method
## [4] print.abbrev*
## [5] print.acf*
## ...
## [87] print.data.frame  <== Here it is!
## [88] print.data.table
## ...
```

6. R now calls the `print.data.frame()`, passing in `df`. The data frame is printed to the console.
7. When `UseMethod()` completes execution, it does not return control to the enclosing function `print()` from which it was called. The work of printing is done, so R arranges for control to be passed back to whomever called `print()` in the first place.

It is interesting to note that the very act of "printing out" the print function, which we did earlier in order to see the code for the function, involved a search for a printing method:

```
print      # this is equivalent to print(print)
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x108da3d40>
## <environment: namespace:base>
```

In the call `print(print)`, R looked at the class of `print()`, and found that it was of class `function`. R then searched for a method called `print.function` and found one. Note that this method give the sort of output to the console that would be helpful to a user:

- the code for the function;
- the location of the function in memory: `0x108da3d40`;
- the environment in which the function was defined (the package **base**).

Things go a little bit differently in the call to `print(lst)`. The class of `lst` is `list`, but when you search the results of `methods(print)` you won't find a `print.list()` method; accordingly R uses a fall-back method called `print.default()`. This is why the console-output for lists looks so "neutral."

14.7.2 Terminology

The `print()` function is an example of a *generic* function. A generic function is simply a function that performs a common task by *dispatching* its input to a particular method-function that is selected on the basis of the class of the input to the generic function. Languages that use generic functions are said to support *generic-function OO*.

In message-oriented OO, objects own their own methods, and the way a task is performed depends on the class of the object that is invoked to perform the task. Generic-function OO, which is most commonly found in languages that support functional programming, puts more stress on functions: the generic function "owns" the methods in the sense that it acts as the central dispatcher, assigning a method function to perform a given task. In a bit of a reversal to message-passing OO, the method selected in generic-function OO depends on the class of the input-object to the generic, not on the generic that was called to perform the task.

We should also mention that R actually has two ways to implement generic-function OO:

- S3 classes;
- S4 classes.

S3 classes were the first to be implemented, and they are the implementation we describe here. S4 classes were a later addition. (They tend to be used by programmers who worry that the rules for formation of S3 classes aren't strict enough.)

14.7.3 Common Generic Functions

There are three very commonly-used generic functions in R:

- `print()`, which we have examined already;
- `summary()`;
- `plot()`.

Each of these generics is associated with a large number of method-functions. This is a great advantage to the casual user of R: one has to know only three R-commands in order to acquire useful information about a wide variety of R-objects.

It is always a good idea to "try out" generic functions on objects you are using. You never know if the authors of R, or of a contributed package you have attached, may have written methods that are precisely tailored to that object.

Here are some example of the versatile, polymorphic behavior of the generic function `summary()`:

```
heights <- df$height # vector of class "numeric"
summary(heights)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   51.00   65.00   68.00   67.99   71.75   79.00
```

```
feelings <- df$weight_feel # has class "factor"
summary(feelings)
```

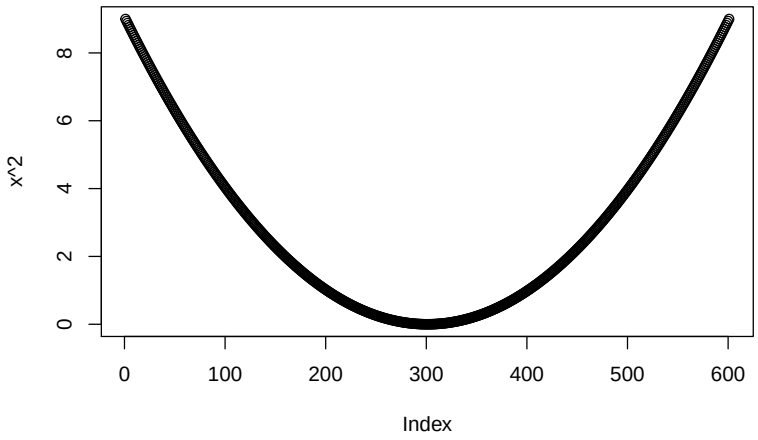


Figure 14.2: Our vector graphed as a parabola!

```
## 1_underweight 2_about_right  3_overweight
##             9            25            37
```

```
summary(df)  # summarizing object of class "data.frame"
```

```
##      height             weight_feel
##  Min.   :51.00   1_underweight: 9
##  1st Qu.:65.00   2_about_right:25
##  Median :68.00   3_overweight :37
##  Mean   :67.99
##  3rd Qu.:71.75
##  Max.   :79.00
```

```
summary(lst)
```

```
##             Length Class  Mode
## height      71     -none- numeric
## weight_feel 71     factor numeric
```

It is interesting also to see how `plot()` reacts to various types of input. See the Figure 14.2

```
x <- seq(-3,3, by = 0.01)
plot(x^2)
```

14.7.4 Writing Your Own Methods

As you advance in your programming skills, you will transition from writing to programs to help you accomplish your own tasks to writing programs that help others—who are not as proficient in programming as you are—get some of *their* work done. Since casual users of R often become accustomed to generic functions as providers of useful information about many types of R-objects, you might find yourself writing methods for one or more of the common generic functions. In this Section will we will practice the art of method-writing: we will write some method-functions to report on the results of a simulation.

Recall the problem from Section 6.6 about estimating the expected number of uniform random numbers one must create until their sum exceeds a specified target-number. Let's rewrite the simulation function so that

it returns an object of a special class. We will then write print and plot methods that permit a user to obtain information about the results of any simulation that was performed.

First of all, let's rewrite `numberNeededSim()`:

```
numberNeededSim <- function(target = 1, reps = 1000,
                            seed = NULL) {

  #set the seed if none is provided
  if ( !is.null(seed) ) {
    set.seed(seed)
  }

  numberNeeded <- function(target) {
    mySum <- 0
    count <- 0
    while( mySum < target ) {
      number <- runif(1)
      mySum <- mySum + number
      count <- count + 1
    }
    count
  }

  needed <- numeric(reps)
  for (i in 1:reps ) {
    needed[i] <- numberNeeded(target)
  }
  results <- list(target = target, sims = needed)
  class(results) <- "numNeededSims"
  results
}
```

In the above code you will note that there is no longer a parameter `table` to permit printing of a table to the console. Also, nothing at all is `cat`-ed to the console. Instead we return only a list with two named elements:

- `target`: the target you want your randomly-generated numbers to sum up to;
- `sims`: the number of numbers required to sum to the target, in each repetition of the simulation.

The class of the returned list is set as "numNeededSims".

Next, we write a print-method function. Its name must be `print.numNeededSims`. All of the table output and `cat`-ing to the console goes here:

```
print.numNeededSims <- function(x) {
  cat("The target was ", x$target, ".\n", sep = "")
  sims <- x$sims
  reps <- length(sims)
  cat("Here is a table of the results, based on ", reps,
      " simulations.\n\n", sep = "")
  tab <- prop.table(table(sims))

  # for sake of pretty output,
  # remove "sims" variable name from top of table printout
```

```
  colNames <- dimnames(tab)
  names(colNames) <- NULL
  dimnames(tab) <- colNames

  print(tab)
  cat("\n")
  cat("The expected number needed is about ",
      mean(sims), ".\n", sep = "")
}
```

Finally, let's write a plot method. Its name must be `plot.numNeededSims`. This method will produce a bar graph of the results of the simulations. We'll use the **ggplot2** plotting package, so we should stop if the user hasn't installed and attached **ggplot2**.

```
plot.numNeededSims <- function(x) {
  if ( !"package:ggplot2" %in% search() ) {
    cat("Need to load package ggplot2 in order to plot.")
  }

  sims <- x$sims
  # for a good bar-plot, convert numerical vector sims
  # to a factor with appropriate levels
  levels <- min(sims):max(sims)
  sims <- factor(sims, levels = levels)

  df <- data.frame(sims)
  plotTitle <- paste0("Results of ", length(sims), " Simulations")
  # in the code below, scale_x_discrete(drop = f) ensures that
  # even if there are no values in sims for a particular level it
  # will still appear in the plot as a zero-height bar
  ggplot(df, aes(x = sims)) + geom_bar() + scale_x_discrete(drop =FALSE) +
    labs(x = "Number Needed", title = plotTitle)
}
```

Let's give it a try:

```
numberNeededSim(reps = 10000, seed = 4040)
```

```
## The target was 1.
## Here is a table of the results, based on 10000 simulations.
##
##      2      3      4      5      6      7
## 0.4974 0.3354 0.1253 0.0339 0.0068 0.0012
##
## The expected number needed is about 2.7209.
```

The print function was called implicitly, so we got useful output to the console.

It's also possible to save the results somewhere, for example:

```
results <- numberNeededSim(reps = 10000, seed = 4040)
str(results)
```

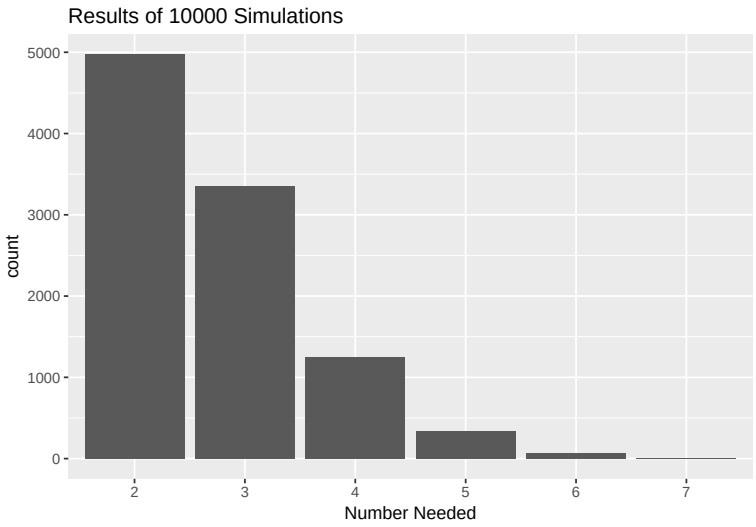


Figure 14.3: Results of the Number-Needed simulation.

```
## List of 2
##  $ target: num 1
##  $ sims  : num [1:10000] 3 2 2 2 3 3 4 2 2 5 ...
##  - attr(*, "class")= chr "numNeededSims"
```

Then it's possible for the user to recall specific features of the results, for example:

```
results$target  # get just the target number
```

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

If we wanted the printout we could just say:

```
results
```

```
## The target was 1.
## Here is a table of the results, based on 10000 simulations.
##
##      2      3      4      5      6      7
## 0.4974 0.3354 0.1253 0.0339 0.0068 0.0012
##
## The expected number needed is about 2.7209.
```

For a plot we can use the `plot()` generic. The resulting graph appears as Figure 14.3.

```
plot(results)
```

14.7.5 Writing a Generic Function

Generic functions are most useful when they correspond to tasks that are so commonly performed that many methods are written for them, so that users get in the habit of "trying out" the generic on their object. As a consequence, the vast majority of method-functions are written for currently-existing, very well-known generics like `print()`, `summary()` and `plot()`. It doesn't make a lot of sense to write generics that will have

only a few methods attached to them. Nevertheless, it's highly instructive to see how generics do their work, so as as example we'll write a new generic, along with a couple of method functions.[3]

First let's create some objects with special classes. Here are two objects of class "cartesianPoint". Our intention is that they correspond to point on the plane, represented with the standard x and y Cartesian coordinates.

```
point1 <- list(x = 3, y = 4)
class(point1) <- "cartesianPoint"
point2 <- list(x = 2, y = 5)
class(point2) <- "cartesianPoint"
```

It is also possible to represent a point on the plan with *polar coordinates*. The elements of a polar coordinates representation are:

- r: a non-negative real number that the distance from the origin to the point;
- θ: the angle measure (in radians) between the positive x-axis and ray from the origin to the point.

```
point3 <- list(r = 2, theta = pi/2)
point4 <- list(r = 1, theta = pi)
class(point3) <- "polarPoint"
class(point4) <- "polarPoint"
```

In the definition above, `point3` is the point that lies at $\pi/2$ radians (90 degrees) counter-clockwise from the positive x-axis. That means that it lies along the positive y-axis. It is 2 units from the origin, so in Cartesian coordinates it would be written as $(0, 2)$. Similarly, `point4` would be written in Cartesian coordinates as $(-1, 0)$, since it lies one unit from the origin along the negative x-axis.

Now let us suppose that we would like to find the x-coordinate of a point. For points of class `cartesianPoint` this is pretty simple:

```
point1$x  # gives x-coordinate
```

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

If the point is given in polar coordinates, we must convert it to Cartesian coordinates. You may recall the conversion formulas from a previous trigonometry class. To get x, use:

$$x = r\cos\theta.$$

To get y, use:

$$x = r\cos\theta.$$

Thus, to find the x-coordinate for `point3`, work as follows:

```
point3$r * cos(point3$theta)
```

```
## [1] 1.224647e-16
```

The result is 0 (to a tiny bit of round-off error).

We now write a generic function `xpos()` for the x-coordinate:

[3]The generic we write is drawn from an example provided in the official *R Languge Definition* (R Core Team (2017)), written by the developers of R.

```
xpos <- function(x) {
  UseMethod("xpos")
}
```

We need to write our method functions, one for each point class:

```
xpos.cartesianPoint <- function(point) {
  point$x
}

xpos.polarPoint <- function(point) {
  point$r * cos(point$theta)
}
```

Now we can feed points of either class into the generic `xpos()` function:

```
xpos(point2)
```

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

```
xpos(point4)
```

```
## [1] -1
```

Glossary

Object-Oriented Programming A programming paradigm in which programs are built around objects, which are complex structures that contain data.

Class A general prototype from which individual objects may be created. The definition of the class specifies the attributes and methods that shall be possessed by any object created from that class. In addition, the definition of the class includes a function called an *initializer* that governs the creation of individual objects from the class.

Instantiation The creation of an individual object as an instance of a class. The object gets all of the attributes and methods of the class (except for the initializer function). Typically the intializer functions allows for determination of the values of some of the object's attributes at the time of instantiation.

Message-Passing OO A type of object-oriented programming in which a task is performed by passing a message to the object that will perform the task. The method by which the object performs the task is determined solely by the class of which the object is an instance.

Attribute A data-field belonging to an object that is not a function.

Method (also called "Method-Function") A function that encapsulates a particular way of performing a task. In message-passing OO, it is a function data-field belonging to an object that as a data-field. Such a function usually has access to its inputs, to other data from its object, and to the objet itself. In generic-function OO, it is a function that is accessed through a generic function.

Reference Semantics When an object has *reference semantics*, assignments involving that object create a pointer to the object, rather than creating a copy of the object itself.

Composition The situation that arises when an object with reference semantics contains one or more other objects with reference semantics as data-fields.

Inheritance The situation that arises when a class (known as the *child class*) is defined as being a particular type of some other class (known as the *parent class*). By default the child class has all of the attributes and methods of the parent class. The child class may be given additional attributes and methods.

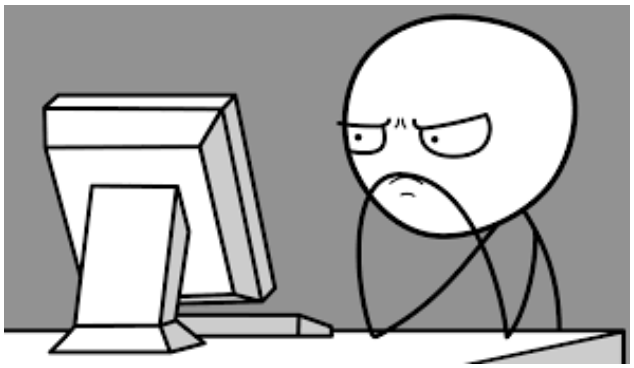
Overriding When a method defined in a child class has the same name as a method belonging to the parent class, then the child-class method is said to *override* the parent-class method. When the method is called on an instance of the child class the defining code in the child class, not the parent class, is used to execute the method.

Generic Function A function that dispatches an input object to one of a number of method-functions, based on the class of the input.

Generic-Function OO A type of object-oriented programming in which tasks are performed by generic functions. The method used to perform a particular task is determined by the class of the input object.

Polymorphism A program exhibits *polymorphism* when a function behaves differently depending either on the type of object to which it belongs or the type of object to which it is applied.

Exercises



1. Write a new class called Witch that inherits from class Person. It should have two additional properties:

   - `shoes`: Color of slippers worn by an individual witch. The initial value should be NULL.
   - `region`: the part of Oz over which an individual witch reigns (e.g., "North", "South", etc.). The initial value should be NULL.

   The class should include set-methods for both `shoes` and `region`. In addition there should be an `initialize()` method that overrides the method already provided with class Person. This new method should permit the user to set the value of `slippers` and `region`.

   Create two new witches:

   - The Wicked Witch of the East. Her name and desire are up to you, but her slippers should be silver, and of course she should reign over the East.
   - Glinda, the Good Witch of the North. Her desire and color of shoes are up to you.

2. Revisit the class Ocean, and rewrite its `initialize()` method so that you can set the age, time to maturity, range and stepSize individually for each new whale. You should be able to use it like this:

```
maleInfo <- data.frame(age = c(10, 12, 13, 8, 7, 14, 2),
                                maturity = rep(10, times = 7),
                                range = c(4, 4, 5, 6, 6, 6, 8),
                                stepSize = c(3,3,3,4,4,5,6))
femaleInfo <- data.frame(age = c(9, 11, 15, 23, 4, 15, 4),
                                maturity = rep(8, times = 7),
                                range = c(7, 4, 3, 6, 9, 6, 8),
                                stepSize = c(3,3,10,2,3,8,5)))
ocean <- Ocean$new(dims = c(100, 100),
                   males = maleInfo,
                   females = femaleInfo)
```

   Run a simulation, initializing the ocean with ten whales of each sex. Their specific characteristics are up to you, as is the value of the starvation parameter. Write an R Markdown document that incorporates the appropriate code and includes a discussion of your results.

3. Recall the function `meetupSim()` from Section 6.4, with which we investigated the probability that Anna and Raj would meet at the Coffee Shop. Suppose that we are interested not only in the probability that they meet, but also in the distribution of the number of minutes by which the latecomer misses the one who cam early on the occasions when they do not manage to connect. Revise `meetupSim()` so that it does not print any results to the console, but instead returns a list. The list should have two elements:

- a logical vector indicating, for each repetition of the simulation, whether or not Anna and Raj met;
- a numerical vector indicating, for each repetition in which they did not meet, the number of minutes by which the latecomer missed meeting the person who arrived earlier.

Make the list have class "meetupSims". A typical example of use should look like this:

```
results <- meetupSim(reps = 10000, seed = 3535)
str(results)
```

```
## List of 2
##  $ connect: logi [1:10000] FALSE FALSE TRUE FALSE TRUE FALSE ...
##  $ noMeet : num [1:6934] 30.275 29.959 22.394 0.491 4.898 ...
##  - attr(*, "class")= chr "meetupSims"
```

4. Building on the previous exercise, write a method-function called `print.meetupSims()` that prints the results of a meet-up simulation to the console. The function should provide a table that gives the proportion of times that Anna and Raj met, and a numerical summary of the simulation results when they did not meet. (You could use the `summary()` function for this.) A typical example of use would look like this:

```
results   # has same effect as print(results)
```

```
## Here is a table of the results, based on 10000 simulations.
##
## Did not connect       Connected
##          0.6934          0.3066
##
## Summary of how many minutes they missed by:
##
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.00131  6.60680 14.55711 16.55834 24.87720 49.30534
```

5. Continuing in the same vein, write a method-function called `plot.meetupSims()` that makes a density plot showing the distribution of the number of minutes by which the latecomer misses the meeting.

6. Write a generic function called `ypos()` that will return the y-coordinate of points of class `cartesianPoint` and `polarPoint`. Of course you will need to write the corresponding method functions, as well.

7. Write a generic function called `norm()` that will return the distance of a point from the origin. The point could be of class `cartesianPoint` or `polarPoint`, so you will need to write the corresponding method functions, as well. (It will be helpful to recall that for a point with Cartesian coordinates (x, y) the distance from the origin is $\sqrt{x^2 + y^2}$.)

Bibliography

Cena, A., Gagolewski, M., Kosinski, M., Potocka, N., and Zogala-Siudem, B. (2014). *TurtleGraphics: Turtle Graphics in R*. R package version 1.0-5.

Chang, W. (2017). *R6: Classes with Reference Semantics*. R package version 2.2.2.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2017). *shiny: Web Application Framework for R*. R package version 1.0.5.

Downey, A. B. (2015). *Think Python*. Green Tea Press, 2nd edition. ISBN 10: 1491939362.

Grolemund, G. (2014). *Hands on Programming with R*. O'Reilly Media. ISBN-13: 978-1-4493-5901-0.

Harold Abelson, G. J. S. and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press. ISBN-13: 978-0262510875.

Ismay, C. (2016). *Getting used to R, RStudio, and R Markdown*.

J.Nahin, P. (2008). *Digital Dice*. Princeton University Press. ISBN-13: 978-0-691-12698-2.

Matloff, N. (2011). *The Art of R Programming*. No Starch Press Press, 2nd edition. ISBN 13: 978-1-59327-384-2.

Papert, S. (1993). *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books. ISBN-13: 978-0465046744.

Pruim, R., Kaplan, D., and Horton, N. (2016). *mosaicData: Project MOSAIC Data Sets*. R package version 0.14.0.

R Core Team (2017). *R Language Definition*. R Foundation for Statistical Computing, Vienna, Austria.

Robinson, R. and White, H. (2016). *tigerstats: R Functions for Elementary Statistics*. R package version 0.3.

White, H. (2017). *tigerData: GC Statistics Datasets*. R package version 0.2.

Wickham, H. (2014). *Advanced R*. Chapman and Hall. ISBN-10: 1466586966.

Wickham, H. (2015). *pryr: Tools for Computing on the Language*. R package version 0.1.2.

Wickham, H. (2016). *plyr: Tools for Splitting, Applying and Combining Data*. R package version 1.8.4.

Wickham, H. and Chang, W. (2017). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. http://ggplot2.tidyverse.org, https://github.com/tidyverse/ggplot2.

Wickham, H., Hester, J., and Francois, R. (2017). *readr: Read Rectangular Text Data*. R package version 1.1.1.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer Verlag. ISBN-13: 978-0-387-28695-2.

Xie, Y. (2017a). *blogdown: Create Blogs and Websites with R Markdown*. R package version 0.1.

Xie, Y. (2017b). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.5.

Index