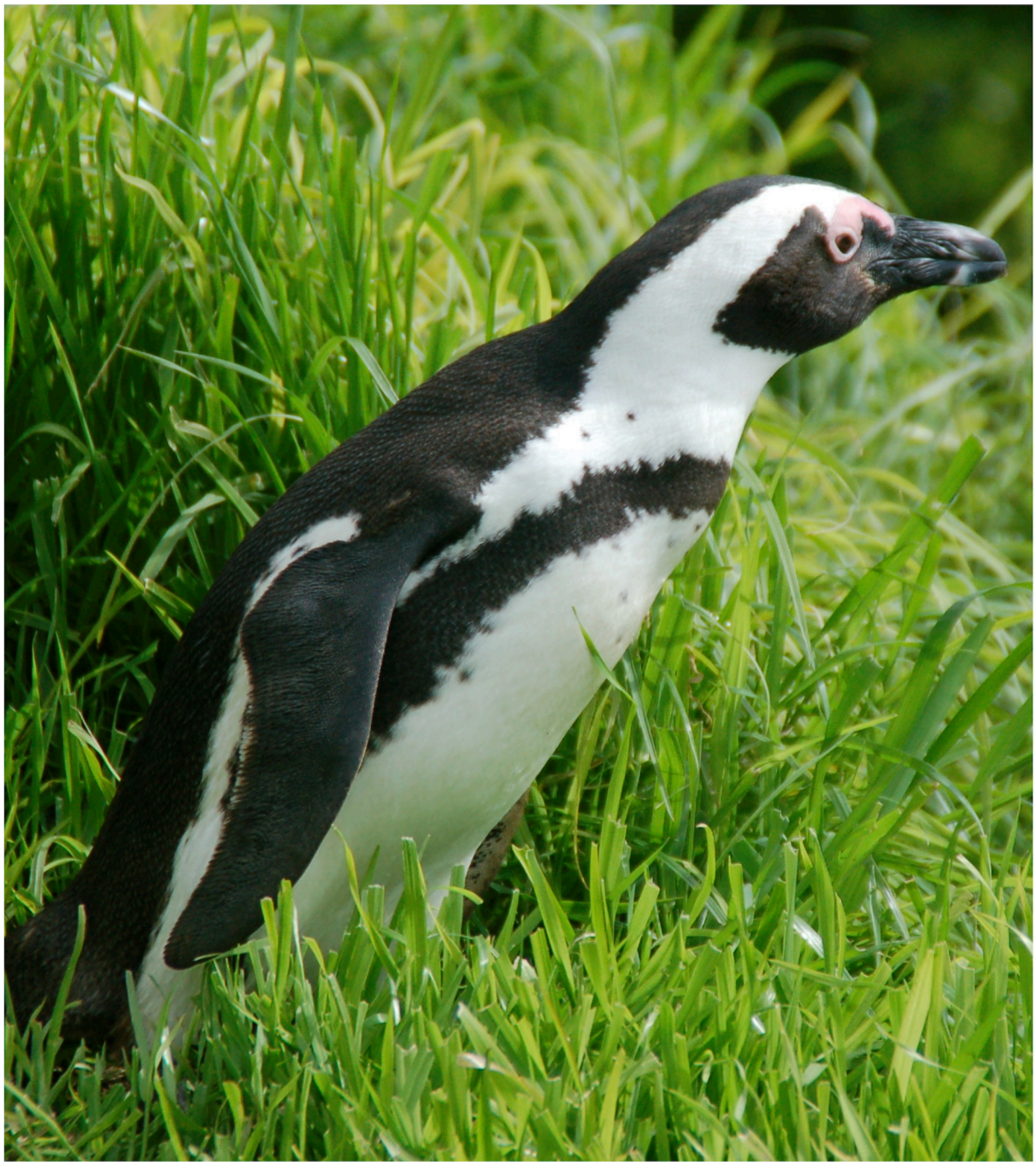
R Programming for Data Science



Roger D. Peng

R Programming for Data Science

Roger D. Peng

This book is for sale at http://leanpub.com/rprogramming

This version was published on 2022-05-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Roger D. Peng

The Art of Data Science

Exploratory Data Analysis with R

Executive Data Science

Report Writing for Data Science in R

Advanced Statistical Computing

The Data Science Salon

Conversations On Data Science

Mastering Software Development in R

Essays on Data Analysis

Tidyverse Skills for Data Science in R

Contents

1. Stay in Touch! **1**

2. Preface **2**

3. History and Overview of R **5**
- 3.1 What is R? 5
- 3.2 What is S? 5
- 3.3 The S Philosophy 6
- 3.4 Back to R 6
- 3.5 Basic Features of R 7
- 3.6 Free Software 7
- 3.7 Design of the R System 8
- 3.8 Limitations of R 9
- 3.9 R Resources 10

4. Getting Started with R **12**
- 4.1 Installation 12
- 4.2 Getting started with the R interface 12

5. R Nuts and Bolts **13**
- 5.1 Entering Input 13
- 5.2 Evaluation 13
- 5.3 R Objects 14
- 5.4 Numbers 15
- 5.5 Attributes 15
- 5.6 Creating Vectors 15
- 5.7 Mixing Objects 16
- 5.8 Explicit Coercion 16
- 5.9 Matrices 17
- 5.10 Lists 18
- 5.11 Factors 19
- 5.12 Missing Values 20
- 5.13 Data Frames 21
- 5.14 Names 22
- 5.15 Summary 23

6. Getting Data In and Out of R . . . 24
6.1 Reading and Writing Data . . . 24
6.2 Reading Data Files with `read.table()` . . . 24
6.3 Reading in Larger Datasets with read.table . . . 25
6.4 Calculating Memory Requirements for R Objects . . . 26

7. Using the `readr` Package . . . 28

8. Using Textual and Binary Formats for Storing Data . . . 32
8.1 Using `dput()` and `dump()` . . . 32
8.2 Binary Formats . . . 34

9. Interfaces to the Outside World . . . 36
9.1 File Connections . . . 36
9.2 Reading Lines of a Text File . . . 37
9.3 Reading From a URL Connection . . . 38

10. Subsetting R Objects . . . 39
10.1 Subsetting a Vector . . . 39
10.2 Subsetting a Matrix . . . 40
10.3 Subsetting Lists . . . 41
10.4 Subsetting Nested Elements of a List . . . 42
10.5 Extracting Multiple Elements of a List . . . 43
10.6 Partial Matching . . . 43
10.7 Removing NA Values . . . 44

11. Vectorized Operations . . . 46
11.1 Vectorized Matrix Operations . . . 47

12. Dates and Times . . . 48
12.1 Dates in R . . . 48
12.2 Times in R . . . 48
12.3 Operations on Dates and Times . . . 50
12.4 Summary . . . 51

13. Managing Data Frames with the `dplyr` package . . . 52
13.1 Data Frames . . . 52
13.2 The `dplyr` Package . . . 52
13.3 `dplyr` Grammar . . . 53
13.4 Installing the `dplyr` package . . . 53
13.5 `select()` . . . 54
13.6 `filter()` . . . 56
13.7 `arrange()` . . . 58
13.8 `rename()` . . . 59
13.9 `mutate()` . . . 59

3. History and Overview of R

> There are only two kinds of languages: the ones people complain about and the ones nobody uses —*Bjarne Stroustrup*

Watch a video of this chapter[1]

3.1 What is R?

This is an easy question to answer. R is a dialect of S.

3.2 What is S?

S is a language that was developed by John Chambers and others at the old Bell Telephone Laboratories, originally part of AT&T Corp. S was initiated in 1976[2] as an internal statistical analysis environment—originally implemented as Fortran libraries. Early versions of the language did not even contain functions for statistical modeling.

In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book *Statistical Models in S* by Chambers and Hastie (the white book) documents the statistical analysis functionality. Version 4 of the S language was released in 1998 and is the version we use today. The book *Programming with Data* by John Chambers (the green book) documents this version of the language.

Since the early 90's the life of the S language has gone down a rather winding path. In 1993 Bell Labs gave StatSci (later Insightful Corp.) an exclusive license to develop and sell the S language. In 2004 Insightful purchased the S language from Lucent for $2 million. In 2006, Alcatel purchased Lucent Technologies and is now called Alcatel-Lucent.

Insightful sold its implementation of the S language under the product name S-PLUS and built a number of fancy features (GUIs, mostly) on top of it—hence the "PLUS". In 2008 Insightful was acquired by TIBCO for $25 million. As of this writing TIBCO is the current owner of the S language and is its exclusive developer.

The fundamentals of the S language itself has not changed dramatically since the publication of the Green Book by John Chambers in 1998. In 1998, S won the Association for Computing Machinery's Software System Award, a highly prestigious award in the computer science field.

[1] https://youtu.be/STihTnVSZnI
[2] http://cm.bell-labs.com/stat/doc/94.11.ps

3.3 The S Philosophy

The general S philosophy is important to understand for users of S and R because it sets the stage for the design of the language itself, which many programming veterans find a bit odd and confusing. In particular, it's important to realize that the S language had its roots in data analysis, and did not come from a traditional programming language background. Its inventors were focused on figuring out how to make data analysis easier, first for themselves, and then eventually for others.

In Stages in the Evolution of S[3], John Chambers writes:

> "[W]e wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important."

The key part here was the transition from *user* to *developer*. They wanted to build a language that could easily service both "people". More technically, they needed to build language that would be suitable for interactive data analysis (more command-line based) as well as for writing longer programs (more traditional programming language-like).

3.4 Back to R

The R language came to use quite a bit after S had been developed. One key limitation of the S language was that it was only available in a commericial package, S-PLUS. In 1991, R was created by Ross Ihaka and Robert Gentleman in the Department of Statistics at the University of Auckland. In 1993 the first announcement of R was made to the public. Ross's and Robert's experience developing R is documented in a 1996 paper in the *Journal of Computational and Graphical Statistics*:

> Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics, 5(3):299–314, 1996

In 1995, Martin Mächler made an important contribution by convincing Ross and Robert to use the GNU General Public License[4] to make R free software. This was critical because it allowed for the source code for the entire R system to be accessible to anyone who wanted to tinker with it (more on free software later).

In 1996, a public mailing list was created (the R-help and R-devel lists) and in 1997 the R Core Group was formed, containing some people associated with S and S-PLUS. Currently, the core group controls the source code for R and is solely able to check in changes to the main R source tree. Finally, in 2000 R version 1.0.0 was released to the public.

[3] http://www.stat.bell-labs.com/S/history.html
[4] http://www.gnu.org/licenses/gpl-2.0.html

3.5 Basic Features of R

In the early days, a key feature of R was that its syntax is very similar to S, making it easy for S-PLUS users to switch over. While the R's syntax is nearly identical to that of S's, R's semantics, while superficially similar to S, are quite different. In fact, R is technically much closer to the Scheme language than it is to the original S language when it comes to how R works under the hood.

Today R runs on almost any standard computing platform and operating system. Its open source nature means that anyone is free to adapt the software to whatever platform they choose. Indeed, R has been reported to be running on modern tablets, phones, PDAs, and game consoles.

One nice feature that R shares with many popular open source projects is frequent releases. These days there is a major annual release, typically in October, where major new features are incorporated and released to the public. Throughout the year, smaller-scale bugfix releases will be made as needed. The frequent releases and regular release cycle indicates active development of the software and ensures that bugs will be addressed in a timely manner. Of course, while the core developers control the primary source tree for R, many people around the world make contributions in the form of new feature, bug fixes, or both.

Another key advantage that R has over many other statistical packages (even today) is its sophisticated graphics capabilities. R's ability to create "publication quality" graphics has existed since the very beginning and has generally been better than competing packages. Today, with many more visualization packages available than before, that trend continues. R's base graphics system allows for very fine control over essentially every aspect of a plot or graph. Other newer graphics systems, like lattice and ggplot2 allow for complex and sophisticated visualizations of high-dimensional data.

R has maintained the original S philosophy, which is that it provides a language that is both useful for interactive work, but contains a powerful programming language for developing new tools. This allows the user, who takes existing tools and applies them to data, to slowly but surely become a developer who is creating new tools.

Finally, one of the joys of using R has nothing to do with the language itself, but rather with the active and vibrant user community. In many ways, a language is successful inasmuch as it creates a platform with which many people can create new things. R is that platform and thousands of people around the world have come together to make contributions to R, to develop packages, and help each other use R for all kinds of applications. The R-help and R-devel mailing lists have been highly active for over a decade now and there is considerable activity on web sites like Stack Overflow.

3.6 Free Software

A major advantage that R has over many other statistical packages and is that it's free in the sense of free software (it's also free in the sense of free beer). The copyright for the primary source code for R is held by the R Foundation[5] and is published under the GNU General Public License version

[5] http://www.r-project.org/foundation/

2.0[6].

According to the Free Software Foundation, with *free software*, you are granted the following four freedoms[7]

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

You can visit the Free Software Foundation's web site[8] to learn a lot more about free software. The Free Software Foundation was founded by Richard Stallman in 1985 and Stallman's personal web site[9] is an interesting read if you happen to have some spare time.

3.7 Design of the R System

The primary R system is available from the Comprehensive R Archive Network[10], also known as CRAN. CRAN also hosts many add-on packages that can be used to extend the functionality of R.

The R system is divided into 2 conceptual parts:

1. The "base" R system that you download from CRAN: Linux[11] Windows[12] Mac[13] Source Code[14]
2. Everything else.

R functionality is divided into a number of *packages*.

- The "base" R system contains, among other things, the `base` package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include `utils`, `stats`, `datasets`, `graphics`, `grDevices`, `grid`, `methods`, `tools`, `parallel`, `compiler`, `splines`, `tcltk`, `stats4`.
- There are also "Recommended" packages: `boot`, `class`, `cluster`, `codetools`, `foreign`, `KernSmooth`, `lattice`, `mgcv`, `nlme`, `rpart`, `survival`, `MASS`, `spatial`, `nnet`, `Matrix`.

[6] http://www.gnu.org/licenses/gpl-2.0.html
[7] http://www.gnu.org/philosophy/free-sw.html
[8] http://www.fsf.org
[9] https://stallman.org
[10] http://cran.r-project.org
[11] http://cran.r-project.org/bin/linux/
[12] http://cran.r-project.org/bin/windows/
[13] http://cran.r-project.org/bin/macosx/
[14] http://cran.r-project.org/src/base/R-3/R-3.1.3.tar.gz

When you download a fresh installation of R from CRAN, you get all of the above, which represents a substantial amount of functionality. However, there are many other packages available:

- There are over 4000 packages on CRAN that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project[15].
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.
- There are a number of packages being developed on repositories like GitHub and BitBucket but there is no reliable listing of all these packages.

3.8 Limitations of R

No programming language or statistical analysis system is perfect. R certainly has a number of drawbacks. For starters, R is essentially based on almost 50 year old technology, going back to the original S system developed at Bell Labs. There was originally little built in support for dynamic or 3-D graphics (but things have improved greatly since the "old days").

Another commonly cited limitation of R is that objects must generally be stored in physical memory. This is in part due to the scoping rules of the language, but R generally is more of a memory hog than other statistical packages. However, there have been a number of advancements to deal with this, both in the R core and also in a number of packages developed by contributors. Also, computing power and capacity has continued to grow over time and amount of physical memory that can be installed on even a consumer-level laptop is substantial. While we will likely never have enough physical memory on a computer to handle the increasingly large datasets that are being generated, the situation has gotten quite a bit easier over time.

At a higher level one "limitation" of R is that its functionality is based on consumer demand and (voluntary) user contributions. If no one feels like implementing your favorite method, then it's *your* job to implement it (or you need to pay someone to do it). The capabilities of the R system generally reflect the interests of the R user community. As the community has ballooned in size over the past 10 years, the capabilities have similarly increased. When I first started using R, there was very little in the way of functionality for the physical sciences (physics, astronomy, etc.). However, now some of those communities have adopted R and we are seeing more code being written for those kinds of applications.

If you want to know my general views on the usefulness of R, you can see them here in the following exchange on the R-help mailing list with Douglas Bates and Brian Ripley in June 2004:

> **Roger D. Peng**: I don't think anyone actually believes that R is designed to make *everyone* happy. For me, R does about 99% of the things I need to do, but sadly, when I need to order a pizza, I still have to pick up the telephone.

[15] http://bioconductor.org

> **Douglas Bates**: There are several chains of pizzerias in the U.S. that provide for Internet-based ordering (e.g. www.papajohnsonline.com) so, with the Internet modules in R, it's only a matter of time before you will have a pizza-ordering function available.
>
> **Brian D. Ripley**: Indeed, the GraphApp toolkit (used for the RGui interface under R for Windows, but Guido forgot to include it) provides one (for use in Sydney, Australia, we presume as that is where the GraphApp author hails from). Alternatively, a Padovian has no need of ordering pizzas with both home and neighbourhood restaurants ….

At this point in time, I think it would be fairly straightforward to build a pizza ordering R package using something like the `RCurl` or `httr` packages. Any takers?

3.9 R Resources

Official Manuals

As far as getting started with R by reading stuff, there is of course this book. Also, available from CRAN[16] are

- An Introduction to R[17]
- R Data Import/Export[18]
- Writing R Extensions[19]: Discusses how to write and organize R packages
- R Installation and Administration[20]: This is mostly for building R from the source code)
- R Internals[21]: This manual describes the low level structure of R and is primarily for developers and R core members
- R Language Definition[22]: This documents the R language and, again, is primarily for developers

Useful Standard Texts on S and R

- Chambers (2008). *Software for Data Analysis*, Springer
- Chambers (1998). *Programming with Data*, Springer: This book is *not* about R, but it describes the organization and philosophy of the current version of the S language, and is a useful reference.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer: This is a standard textbook in statistics and describes how to use many statistical methods in R. This book has an associated R package (the `MASS` package) that comes with every installation of R.

[16] http://cran.r-project.org
[17] http://cran.r-project.org/doc/manuals/r-release/R-intro.html
[18] http://cran.r-project.org/doc/manuals/r-release/R-data.html
[19] http://cran.r-project.org/doc/manuals/r-release/R-exts.html
[20] http://cran.r-project.org/doc/manuals/r-release/R-admin.html
[21] http://cran.r-project.org/doc/manuals/r-release/R-ints.html
[22] http://cran.r-project.org/doc/manuals/r-release/R-lang.html

- Venables & Ripley (2000). *S Programming*, Springer: This book is a little old but is still relevant and accurate. Despite its title, this book is useful for R also.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press: Paul Murrell wrote and designed much of the graphics system in R and this book essentially documents the underlying details. This is not so much a "user-level" book as a developer-level book. But it is an important book for anyone interested in designing new types of graphics or visualizations.
- Wickham (2014). *Advanced R*, Chapman & Hall/CRC Press: This book by Hadley Wickham covers a number of areas including object-oriented programming, functional programming, profiling and other advanced topics.

Other Resources

- Major technical publishers like Springer, Chapman & Hall/CRC have entire series of books dedicated to using R in various applications. For example, Springer has a series of books called *Use R!*.
- A longer list of books can be found on the CRAN web site[23].

[23] http://www.r-project.org/doc/bib/R-books.html

4. Getting Started with R

4.1 Installation

The first thing you need to do to get started with R is to install it on your computer. R works on pretty much every platform available, including the widely available Windows, Mac OS X, and Linux systems. If you want to watch a step-by-step tutorial on how to install R for Mac or Windows, you can watch these videos:

- Installing R on Windows[1]
- Installing R on the Mac[2]

There is also an integrated development environment available for R that is built by RStudio. I really like this IDE—it has a nice editor with syntax highlighting, there is an R object viewer, and there are a number of other nice features that are integrated. You can see how to install RStudio here

- Installing RStudio[3]

The RStudio IDE is available from RStudio's web site[4].

4.2 Getting started with the R interface

After you install R you will need to launch it and start writing R code. Before we get to exactly how to write R code, it's useful to get a sense of how the system is organized. In these two videos I talk about where to write code and how set your working directory, which let's R know where to find all of your files.

- Writing code and setting your working directory on the Mac[5]
- Writing code and setting your working directory on Windows[6]

[1] http://youtu.be/Ohnk9hcxf9M
[2] https://youtu.be/uxuuWXU-7UQ
[3] https://youtu.be/bM7Sfz-LADM
[4] http://rstudio.com
[5] https://youtu.be/8xT3hmJQskU
[6] https://youtu.be/XBcvH1BpIBo

5. R Nuts and Bolts

5.1 Entering Input

Watch a video of this section[1]

At the R prompt we type expressions. The `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <-  ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

5.2 Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5  ## nothing printed
> x       ## auto-printing occurs
[1] 5
> print(x)  ## explicit printing
[1] 5
```

The `[1]` shown in the output indicates that `x` is a vector and `5` is its first element.

Typically with interactive work, we do not explicitly print objects with the `print` function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However,

[1] https://youtu.be/vGY5i_J2c-c?t=4m43s

when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.

When an R vector is printed you will notice that an index for the vector is printed in square brackets `[]` on the side. For example, see this integer sequence of length 20.

```
> x <- 11:30
> x
 [1] 11 12 13 14 15 16 17 18 19 20 21 22
[13] 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself, they are merely part of the *printed output*.

With R, it's important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object.

Note that the `:` operator is used to create integer sequences.

5.3 R Objects

Watch a video of this section[2]

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the `vector()` function. There is really only one rule about vectors in R, which is that **A vector can only contain objects of the same class**.

But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that's usually why we use them.

There is also a class for "raw" objects, but they are not commonly used directly in data analysis and I won't cover them here.

[2] https://youtu.be/vGY5i_J2c-c

5.4 Numbers

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like "1" or "2" in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like "1.00" or "2.00"). This isn't important most of the time...except when it is.

If you explicitly want an integer, you need to specify the `L` suffix. So entering `1` in R gives you a numeric object; entering `1L` explicitly gives you an integer object.

There is also a special number `Inf` which represents infinity. This allows us to represent entities like `1 / 0`. This way, `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0.

The value `NaN` represents an undefined value ("not a number"); e.g. `0 / 0`; `NaN` can also be thought of as a missing value (more on that later)

5.5 Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the `attributes()` function. Not all R objects contain attributes, in which case the `attributes()` function returns `NULL`.

5.6 Creating Vectors

Watch a video of this section[3]

The `c()` function can be used to create vectors of objects by concatenating things together.

[3] https://youtu.be/w8_XdYI3reU

```
> x <- c(0.5, 0.6)       ## numeric
> x <- c(TRUE, FALSE)    ## logical
> x <- c(T, F)           ## logical
> x <- c("a", "b", "c")  ## character
> x <- 9:29              ## integer
> x <- c(1+0i, 2+4i)     ## complex
```

Note that in the above example, `T` and `F` are short-hand ways to specify `TRUE` and `FALSE`. However, in general one should try to use the explicit `TRUE` and `FALSE` values when indicating logical values. The `T` and `F` values are primarily there for when you're feeling lazy.

You can also use the `vector()` function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

5.7 Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

In the example above, we see the effect of *implicit coercion*. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and...sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

5.8 Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in `NA`s being produced.

```
> x <- c("a", "b", "c")
> as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA
> as.logical(x)
[1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

5.9 Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices can be created by *column-binding* or *row-binding* with the `cbind()` and `rbind()` functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

5.10 Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination.

Lists can be explicitly created using the `list()` function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the `vector()` function

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL

[[5]]
NULL
```

5.11 Factors

Watch a video of this section[4]

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modeling and are treated specially by modelling functions like `lm()` and `glm()`.

Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factor objects can be created with the `factor()` function.

[4] https://youtu.be/NuY6jY4qE7I

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no
Levels: no yes
> table(x)
x
 no yes
  2   3
> ## See the underlying representation of factor
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like `read.table()`. Those functions often default to creating factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x  ## Levels are put in alphabetical order
[1] yes yes no  yes no
Levels: no yes
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+             levels = c("yes", "no"))
> x
[1] yes yes no  yes no
Levels: yes no
```

5.12 Missing Values

Missing values are denoted by `NA` or `NaN` for q undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE


> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

5.13 Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr[5] has an optimized set of functions designed to work efficiently with data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called `row.names` which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the `read.table()` or `read.csv()`. However, data frames can also be created explicitly with the `data.frame()` function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling `data.matrix()`. While it might seem that the `as.matrix()` function should be used to coerce a data frame to a matrix, almost always, what you want is the result of `data.matrix()`.

[5] https://github.com/hadley/dplyr

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

5.14 Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
   New York     Seattle Los Angeles
          1           2           3
> names(x)
[1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los Angeles`
[1] 1

$Boston
[1] 2

$London
[1] 3
> names(x)
[1] "Los Angeles" "Boston"      "London"
```

Matrices can have both column and row names.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the `colnames()` and `rownames()` functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
  h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the `row.names()` function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the `names()` function. Yes, I know its confusing. Here's a quick summary:

Object	Set column names	Set row names
data frame	`names()`	`row.names()`
matrix	`colnames()`	`rownames()`

5.15 Summary

There are a variety of different builtin-data types in R. In this chapter we have reviewed the following

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames and matrices

All R objects can have attributes that help to describe what is in the object. Perhaps the most useful attribute is names, such as column and row names in a data frame, or simply names in a vector or list. Attributes like dimensions are also important as they can modify the behavior of objects, like turning a vector into a matrix.

6. Getting Data In and Out of R

6.1 Reading and Writing Data

Watch a video of this section[1]

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

There are analogous functions for writing data to files

- `write.table`, for writing tabular data to text files (i.e. CSV) or connections
- `writeLines`, for writing character data line-by-line to a file or connection
- `dump`, for dumping a textual representation of multiple R objects
- `dput`, for outputting a textual representation of an R object
- `save`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `serialize`, for converting an R object into a binary format for outputting to a connection (or file).

6.2 Reading Data Files with `read.table()`

The `read.table()` function is one of the most commonly used functions for reading data. The help file for `read.table()` is worth reading in its entirety if only because the function gets used a lot (run `?read.table` in R). I know, I know, everyone always says to read the help file, but this one is actually worth reading.

The `read.table()` function has a few important arguments:

[1] https://youtu.be/Z_dc_FADyi4

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset. By default `read.table()` reads an entire file.
- `comment.char`, a character string indicating the comment character. This defalts to `"#"`. If there are no commented lines in your file, it's worth setting this to be the empty string `""`.
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors? This defaults to `TRUE` because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be `FALSE` in those cases. If you *always* want this to be `FALSE`, you can set a global option via `options(stringsAsFactors = FALSE)`. I've never seen so much heat generated on discussion forums about an R function argument than the `stringsAsFactors` argument. Seriously.

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
> data <- read.table("foo.txt")
```

In this case, R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.

Telling R all these things directly makes R run faster and more efficiently. The `read.csv()` function is identical to read.table except that some of the defaults are set differently (like the `sep` argument).

6.3 Reading in Larger Datasets with read.table

Watch a video of this section[2]

With much larger datasets, there are a few things that you can do that will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.

[2] https://youtu.be/BJYYIJO3UFI

- Set `comment.char = ""` if there are no commented lines in your file.
- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

In general, when using R with larger datasets, it's also useful to know a few things about your system.

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- What operating system ar you using? Some operating systems can limit the amount of memory a single process can access

6.4 Calculating Memory Requirements for R Objects

Because R stores all of its objects physical memory, it is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace. One situation where it's particularly important to understand memory requirements is when you are reading in a new dataset into R. Fortunately, it's easy to make a back of the envelope calculation of how much memory will be required by a new dataset.

For example, suppose I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame? Well, on most modern computers double precision floating point numbers[3] are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

[3] http://en.wikipedia.org/wiki/Double-precision_floating-point_format

$$
\begin{aligned}
1{,}500{,}000 \times 120 \times 8 \text{ bytes/numeric} &= 1{,}440{,}000{,}000 \text{ bytes} \\
&= 1{,}440{,}000{,}000 \,/\, 2^{20} \text{ bytes/MB} \\
&= 1{,}373.29 \text{ MB} \\
&= 1.34 \text{ GB}
\end{aligned}
$$

So the dataset would require about 1.34 GB of RAM. Most computers these days have at least that much RAM. However, you need to be aware of

- what other programs might be running on your computer, using up RAM
- what other R objects might already be taking up RAM in your workspace

Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session). This is usually an unpleasant experience that usually requires you to kill the R process, in the best case scenario, or reboot your computer, in the worst case. So make sure to do a rough calculation of memeory requirements before reading in a large dataset. You'll thank me later.

7. Using the `readr` Package

The `readr` package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like `read.table()` and `read.csv()`. The analogous functions in `readr` are `read_table()` and `read_csv()`. These functions are often *much* faster than their base R analogues and provide a few other nice features such as progress meters.

For the most part, you can read use `read_table()` and `read_csv()` pretty much anywhere you might use `read.table()` and `read.csv()`. In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

The importance of the `read_csv` function is perhaps better understood from an historical perspective. R's built in `read.csv` function similarly reads CSV files, but the `read_csv` function in `readr` builds on that by removing some of the quirks and "gotchas" of `read.csv` as well as dramatically optimizing the speed with which it can read data into R. The `read_csv` function also adds some nice user-oriented features like a progress meter and a compact method for specifying column types.

A typical call to `read_csv` will look as follows.

```
> library(readr)
> teams <- read_csv("data/team_standings.csv")
Rows: 32 Columns: 2
── Column specification ────────────────────────────────────────────────────────\
────────────────────────────────────────────────────────

Delimiter: ","
chr (1): Team
dbl (1): Standing

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
> teams
# A tibble: 32 × 2
   Standing Team
      <dbl> <chr>
 1        1 Spain
 2        2 Netherlands
 3        3 Germany
 4        4 Uruguay
 5        5 Argentina
 6        6 Brazil
 7        7 Ghana
```

```
 8        8 Paraguay
 9        9 Japan
10       10 Chile
# … with 22 more rows
```

By default, `read_csv` will open a CSV file and read it in line-by-line. It will also (by default), read in the first few rows of the table in order to figure out the type of each column (i.e. integer, character, etc.). From the `read_csv` help page:

> If 'NULL', all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to supply the correct types yourself.

You can specify the type of each column with the `col_types` argument.

In general, it's a good idea to specify the column types explicitly. This rules out any possible guessing errors on the part of `read_csv`. Also, specifying the column types explicitly provides a useful safety check in case anything about the dataset should change without you knowing about it.

```
> teams <- read_csv("data/team_standings.csv", col_types = "cc")
```

Note that the `col_types` argument accepts a compact representation. Here `"cc"` indicates that the first column is `character` and the second column is `character` (there are only two columns). Using the `col_types` argument is useful because often it is not easy to automatically figure out the type of a column by looking at a few rows (especially if a column has many missing values).

The `read_csv` function will also read compressed files automatically. There is no need to decompress the file first or use the `gzfile` connection function. The following call reads a gzip-compressed CSV file containing download logs from the RStudio CRAN mirror.

```
> logs <- read_csv("data/2016-07-19.csv.bz2", n_max = 10)
Rows: 10 Columns: 10
── Column specification ────────────────────────────────────────────────────────\
────────────────────────────────────────────────────
Delimiter: ","
chr  (6): r_version, r_arch, r_os, package, version, country
dbl  (2): size, ip_id
date (1): date
time (1): time

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Note that the warnings indicate that `read_csv` may have had some difficulty identifying the type of each column. This can be solved by using the `col_types` argument.

```
> logs <- read_csv("data/2016-07-19.csv.bz2", col_types = "ccicccccci", n_max = 10)
> logs
# A tibble: 10 × 10
   date       time     size r_version r_arch r_os  package version country ip_id
   <chr>      <chr>   <int> <chr>     <chr>  <chr> <chr>   <chr>   <chr>   <int>
 1 2016-07-19 22:00… 1.89e6 3.3.0     x86_64 ming… data.t… 1.9.6   US          1
 2 2016-07-19 22:00… 4.54e4 3.3.1     x86_64 ming… assert… 0.1     US          2
 3 2016-07-19 22:00… 1.43e7 3.3.1     x86_64 ming… stringi 1.1.1   DE          3
 4 2016-07-19 22:00… 1.89e6 3.3.1     x86_64 ming… data.t… 1.9.6   US          4
 5 2016-07-19 22:00… 3.90e5 3.3.1     x86_64 ming… foreach 1.4.3   US          4
 6 2016-07-19 22:00… 4.88e4 3.3.1     x86_64 linu… tree    1.0-37  CO          5
 7 2016-07-19 22:00… 5.25e2 3.3.1     x86_64 darw… surviv… 2.39-5  US          6
 8 2016-07-19 22:00… 3.23e6 3.3.1     x86_64 ming… Rcpp    0.12.5  US          2
 9 2016-07-19 22:00… 5.56e5 3.3.1     x86_64 ming… tibble  1.1     US          2
10 2016-07-19 22:00… 1.52e5 3.3.1     x86_64 ming… magrit… 1.5     US          2
```

You can specify the column type in a more detailed fashion by using the various `col_*` functions. For example, in the log data above, the first column is actually a date, so it might make more sense to read it in as a Date variable. If we wanted to just read in that first column, we could do

```
> logdates <- read_csv("data/2016-07-19.csv.bz2",
+                      col_types = cols_only(date = col_date()),
+                      n_max = 10)
> logdates
# A tibble: 10 × 1
   date
   <date>
 1 2016-07-19
 2 2016-07-19
 3 2016-07-19
 4 2016-07-19
 5 2016-07-19
 6 2016-07-19
 7 2016-07-19
 8 2016-07-19
 9 2016-07-19
10 2016-07-19
```

Now the `date` column is stored as a `Date` object which can be used for relevant date-related computations (for example, see the `lubridate` package).

> The `read_csv` function has a `progress` option that defaults to TRUE. This options provides a nice progress meter while the CSV file is being read. However, if you are using `read_csv` in a function,

or perhaps embedding it in a loop, it's probably best to set `progress = FALSE`.