Advanced R

Chapman & Hall/CRC
The R Series

Series Editors

John M. Chambers
Department of Statistics
Stanford University
Stanford, California, USA

Torsten Hothorn
Division of Biostatistics
University of Zurich
Switzerland

Duncan Temple Lang
Department of Statistics
University of California, Davis
Davis, California, USA

Hadley Wickham
RStudio
Boston, Massachusetts, USA

Aims and Scope

This book series reflects the recent rapid growth in the development and application of R, the programming language and software environment for statistical computing and graphics. R is now widely used in academic research, education, and industry. It is constantly growing, with new versions of the core software released regularly and more than 5,000 packages available. It is difficult for the documentation to keep pace with the expansion of the software, and this vital book series provides a forum for the publication of books covering many aspects of the development and application of R.

The scope of the series is wide, covering three main threads:

- Applications of R to specific disciplines such as biology, epidemiology, genetics, engineering, finance, and the social sciences.
- Using R for the study of topics of statistical methodology, such as linear and mixed modeling, time series, Bayesian methods, and missing data.
- The development of R, including programming, building packages, and graphics.

The books will appeal to programmers and developers of R software, as well as applied statisticians and data analysts in many fields. The books will feature detailed worked examples and R code fully integrated into the text, ensuring their usefulness to researchers, practitioners and students.

Published Titles

Stated Preference Methods Using R, *Hideo Aizaki, Tomoaki Nakatani, and Kazuo Sato*

Using R for Numerical Analysis in Science and Engineering, *Victor A. Bloomfield*

Event History Analysis with R, *Göran Broström*

Computational Actuarial Science with R, *Arthur Charpentier*

Statistical Computing in C++ and R, *Randall L. Eubank and Ana Kupresanin*

Reproducible Research with R and RStudio, *Christopher Gandrud*

Introduction to Scientific Programming and Simulation Using R, Second Edition, *Owen Jones, Robert Maillardet, and Andrew Robinson*

Nonparametric Statistical Methods Using R, *John Kloke and Joseph McKean*

Displaying Time Series, Spatial, and Space-Time Data with R, *Oscar Perpiñán Lamigueiro*

Programming Graphical User Interfaces with R, *Michael F. Lawrence and John Verzani*

Analyzing Sensory Data with R, *Sébastien Lê and Theirry Worch*

Analyzing Baseball Data with R, *Max Marchi and Jim Albert*

Growth Curve Analysis and Visualization Using R, *Daniel Mirman*

R Graphics, Second Edition, *Paul Murrell*

Multiple Factor Analysis by Example Using R, *Jérôme Pagès*

Customer and Business Analytics: Applied Data Mining for Business Decision Making Using R, *Daniel S. Putler and Robert E. Krider*

Implementing Reproducible Research, *Victoria Stodden, Friedrich Leisch, and Roger D. Peng*

Using R for Introductory Statistics, Second Edition, *John Verzani*

Advanced R, *Hadley Wickham*

Dynamic Documents with R and knitr, *Yihui Xie*

Advanced R

Hadley Wickham

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20150316

International Standard Book Number-13: 978-1-4665-8696-3 (Paperback)

To Jeff, who makes me happy, and who made sure I had a life outside this book.

Contents

1

Introduction

With more than 10 years experience programming in R, I've had the luxury of being able to spend a lot of time trying to figure out and understand how the language works. This book is my attempt to pass on what I've learned so that you can quickly become an effective R programmer. Reading it will help you avoid the mistakes I've made and dead ends I've gone down, and will teach you useful tools, techniques, and idioms that can help you to attack many types of problems. In the process, I hope to show that, despite its frustrating quirks, R is, at its heart, an elegant and beautiful language, well tailored for data analysis and statistics.

If you are new to R, you might wonder what makes learning such a quirky language worthwhile. To me, some of the best features are:

- It's free, open source, and available on every major platform. As a result, if you do your analysis in R, anyone can easily replicate it.
- A massive set of packages for statistical modelling, machine learning, visualisation, and importing and manipulating data. Whatever model or graphic you're trying to do, chances are that someone has already tried to do it. At a minimum, you can learn from their efforts.
- Cutting edge tools. Researchers in statistics and machine learning will often publish an R package to accompany their articles. This means immediate access to the very latest statistical techniques and implementations.
- Deep-seated language support for data analysis. This includes features likes missing values, data frames, and subsetting.
- A fantastic community. It is easy to get help from experts on the R-help mailing list (`https://stat.ethz.ch/mailman/listinfo/r-help`), stackoverflow (`http://stackoverflow.com/questions/tagged/r`), or subject-specific mailing lists like R-SIG-mixed-models (`https:`

//stat.ethz.ch/mailman/listinfo/r-sig-mixed-models) or ggplot2 (https://groups.google.com/forum/#!forum/ggplot2). You can also connect with other R learners via twitter (https://twitter.com/search?q=%23rstats), linkedin (http://www.linkedin.com/groups/R-Project-Statistical-Computing-77616), and through many local user groups (http://blog.revolutionanalytics.com/local-r-groups.html).

- Powerful tools for communicating your results. R packages make it easy to produce html or pdf reports (http://yihui.name/knitr/), or create interactive websites (http://www.rstudio.com/shiny/).

- A strong foundation in functional programming. The ideas of functional programming are well suited to solving many of the challenges of data analysis. R provides a powerful and flexible toolkit which allows you to write concise yet descriptive code.

- An IDE (http://www.rstudio.com/ide/) tailored to the needs of interactive data analysis and statistical programming.

- Powerful metaprogramming facilities. R is not just a programming language, it is also an environment for interactive data analysis. Its metaprogramming capabilities allow you to write magically succinct and concise functions and provide an excellent environment for designing domain-specific languages.

- Designed to connect to high-performance programming languages like C, Fortran, and C++.

Of course, R is not perfect. R's biggest challenge is that most R users are not programmers. This means that:

- Much of the R code you'll see in the wild is written in haste to solve a pressing problem. As a result, code is not very elegant, fast, or easy to understand. Most users do not revise their code to address these shortcomings.

- Compared to other programming languages, the R community tends to be more focussed on results instead of processes. Knowledge of software engineering best practices is patchy: for instance, not enough R programmers use source code control or automated testing.

- Metaprogramming is a double-edged sword. Too many R functions use tricks to reduce the amount of typing at the cost of making code that is hard to understand and that can fail in unexpected ways.

- Inconsistency is rife across contributed packages, even within base R. You are confronted with over 20 years of evolution every time you use R. Learning R can be tough because there are many special cases to remember.
- R is not a particularly fast programming language, and poorly written R code can be terribly slow. R is also a profligate user of memory.

Personally, I think these challenges create a great opportunity for experienced programmers to have a profound positive impact on R and the R community. R users do care about writing high quality code, particularly for reproducible research, but they don't yet have the skills to do so. I hope this book will not only help more R users to become R programmers but also encourage programmers from other languages to contribute to R.

1.1 Who should read this book

This book is aimed at two complementary audiences:

- Intermediate R programmers who want to dive deeper into R and learn new strategies for solving diverse problems.
- Programmers from other languages who are learning R and want to understand why R works the way it does.

To get the most out of this book, you'll need to have written a decent amount of code in R or another programming language. You might not know all the details, but you should be familiar with how functions work in R and although you may currently struggle to use them effectively, you should be familiar with the apply family (like `apply()` and `lapply()`).

1.2 What you will get out of this book

This book describes the skills I think an advanced R programmer should have: the ability to produce quality code that can be used in a wide variety of circumstances.

After reading this book, you will:

- Be familiar with the fundamentals of R. You will understand complex data types and the best ways to perform operations on them. You will have a deep understanding of how functions work, and be able to recognise and use the four object systems in R.
- Understand what functional programming means, and why it is a useful tool for data analysis. You'll be able to quickly learn how to use existing tools, and have the knowledge to create your own functional tools when needed.
- Appreciate the double-edged sword of metaprogramming. You'll be able to create functions that use non-standard evaluation in a principled way, saving typing and creating elegant code to express important operations. You'll also understand the dangers of metaprogramming and why you should be careful about its use.
- Have a good intuition for which operations in R are slow or use a lot of memory. You'll know how to use profiling to pinpoint performance bottlenecks, and you'll know enough C++ to convert slow R functions to fast C++ equivalents.
- Be comfortable reading and understanding the majority of R code. You'll recognise common idioms (even if you wouldn't use them yourself) and be able to critique others' code.

1.3 Meta-techniques

There are two meta-techniques that are tremendously helpful for improving your skills as an R programmer: reading source code and adopting a scientific mindset.

Reading source code is important because it will help you write better code. A great place to start developing this skill is to look at the source code of the functions and packages you use most often. You'll find things that are worth emulating in your own code and you'll develop a sense of taste for what makes good R code. You will also see things that you don't like, either because its virtues are not obvious or it offends your sensibilities. Such code is nonetheless valuable, because it helps make concrete your opinions on good and bad code.

A scientific mindset is extremely helpful when learning R. If you don't understand how something works, develop a hypothesis, design some experiments, run them, and record the results. This exercise is extremely useful since if you can't figure something out and need to get help, you can easily show others what you tried. Also, when you learn the right answer, you'll be mentally prepared to update your world view. When I clearly describe a problem to someone else (the art of creating a reproducible example (`http://stackoverflow.com/questions/5963269`)), I often figure out the solution myself.

1.4 Recommended reading

R is still a relatively young language, and the resources to help you understand it are still maturing. In my personal journey to understand R, I've found it particularly helpful to use resources from other programming languages. R has aspects of both functional and object-oriented (OO) programming languages. Learning how these concepts are expressed in R will help you leverage your existing knowledge of other programming languages, and will help you identify areas where you can improve.

To understand why R's object systems work the way they do, I found *The Structure and Interpretation of Computer Programs* (`http://mitpress.mit.edu/sicp/full-text/book/book.html`) (SICP) by Harold Abelson and Gerald Jay Sussman, particularly helpful. It's a concise but deep book. After reading it, I felt for the first time that I could actually design my own object-oriented system. The book was my first introduction to the generic function style of OO common in R. It helped me understand its strengths and weaknesses. SICP also talks a lot about functional programming, and how to create simple functions which become powerful when combined.

To understand the trade-offs that R has made compared to other programming languages, I found *Concepts, Techniques and Models of Computer Programming* (`http://amzn.com/0262220695?tag=devtools-20`) by Peter van Roy and Sef Haridi extremely helpful. It helped me understand that R's copy-on-modify semantics make it substantially easier to reason about code, and that while its current implementation is not particularly efficient, it is a solvable problem.

If you want to learn to be a better programmer, there's no place better

to turn than *The Pragmatic Programmer* (`http://amzn.com/020161622X?tag=devtools-20`) by Andrew Hunt and David Thomas. This book is language agnostic, and provides great advice for how to be a better programmer.

1.5 Getting help

Currently, there are two main venues to get help when you're stuck and can't figure out what's causing the problem: stackoverflow (`http://stackoverflow.com`) and the R-help mailing list. You can get fantastic help in both venues, but they do have their own cultures and expectations. It's usually a good idea to spend a little time lurking, learning about community expectations, before you put up your first post.

Some good general advice:

- Make sure you have the latest version of R and of the package (or packages) you are having problems with. It may be that your problem is the result of a recently fixed bug.
- Spend some time creating a reproducible example (`http://stackoverflow.com/questions/5963269`). This is often a useful process in its own right, because in the course of making the problem reproducible you often figure out what's causing the problem.
- Look for related problems before posting. If someone has already asked your question and it has been answered, it's much faster for everyone if you use the existing answer.

1.6 Acknowledgments

I would like to thank the tireless contributors to R-help and, more recently, stackoverflow (`http://stackoverflow.com/questions/tagged/r`). There are too many to name individually, but I'd particularly like to thank Luke Tierney, John Chambers, Dirk Eddelbuettel, JJ Allaire

and Brian Ripley for generously giving their time and correcting my countless misunderstandings.

This book was written in the open (`https://github.com/hadley/adv-r/`), and chapters were advertised on twitter (`https://twitter.com/hadleywickham`) when complete. It is truly a community effort: many people read drafts, fixed typos, suggested improvements, and contributed content. Without those contributors, the book wouldn't be nearly as good as it is, and I'm deeply grateful for their help. Special thanks go to Peter Li, who read the book from cover-to-cover and provided many fixes. Other outstanding contributors were Aaron Schumacher, @crtahlin, Lingbing Feng, @juancentro, and @johnbaums.

Thanks go to all contributers in alphabetical order: Aaron Schumacher, Aaron Wolen, @aaronwolen, @absolutelyNoWarranty, Adam Hunt, @agrabovsky, @ajdm, @alexbbrown, @alko989, @allegretto, @AmeliaMN, @andrewla, Andy Teucher, Anthony Damico, Anton Antonov, @aranlunzer, @arilamstein, @avilella, @baptiste, @blindjesse, @blmoore, @bnjmn, Brandon Hurr, @BrianDiggs, @Bryce, C. Jason Liang, @Carson, @cdrv, Ching Boon, @chiphogg, Christopher Brown, @christophergandrud, Clay Ford, @cornelius1729, @cplouffe, Craig Citro, @crossfitAL, @crowding, Crt Ahlin, @crtahlin, @cscheid, @csgillespie, @cusanovich, @cwarden, @cwickham, Daniel Lee, @darrkj, @Dasonk, David Hajage, David LeBauer, @dchudz, dennis feehan, @dfeehan, Dirk Eddelbuettel, @dkahle, @dlebauer, @dlschweizer, @dmontaner, @dougmitarotonda, @dpatschke, @duncandonutz, @EdFineOKL, @EDiLD, @eipi10, @elegrand, @EmilRehnberg, Eric C. Anderson, @etb, @fabian-s, Facundo Muñoz, @flammy0530, @fpepin, Frank Farach, @freezby, @fyears, Garrett Grolemund, @garrettgman, @gavinsimpson, @gggtest, Gökçen Eraslan, Gregg Whitworth, @gregorp, @gsee, @gsk3, @gthb, @hassaad85, @i, Iain Dillingham, @ijlyttle, Ilan Man, @imanuelcostigan, @initdch, Jason Asher, Jason Knight, @jasondavies, @jastingo, @jcborras, Jeff Allen, @jeharmse, @jentjr, @JestonBlu, @JimInNashville, @jinlong25, JJ Allaire, Jochen Van de Velde, Johann Hibschman, John Blischak, John Verzani, @johnbaums, @johnjosephhorton, Joris Muller, Joseph Casillas, @juancentro, @kdauria, @kenahoo, @kent37, Kevin Markham, Kevin Ushey, @kforner, Kirill Müller, Kun Ren, Laurent Gatto, @Lawrence-Liu, @ldfmrails, @lgatto, @liangcj, Lingbing Feng, @lynaghk, Maarten Kruijver, Mamoun Benghezal, @mannyishere, Matt Pettis, @mattbaggott, Matthew Grogan, @mattmalin, Michael Kane, @michaelbach, @mjsduncan, @Mullefa, @myqlarson, Nacho Caballero, Nick Carchedi, @nstjhp, @ogennadi, Oliver Keyes, @otepoti, Parker Abercrombie, @patperu,

Patrick Miller, @pdb61, @pengyu, Peter F Schulam, Peter Lindbrook, Peter Meilstrup, @philchalmers, @picasa, @piccolbo, @pierreroudier, @pooryorick, R. Mark Sharp, Ramnath Vaidyanathan, @ramnathv, @Rappster, Ricardo Pietrobon, Richard Cotton, @richardreeve, @rmflight, @rmsharp, Robert M Flight, @RobertZK, @robiRagan, Romain François, @rrunner, @rubenfcasal, @sailingwave, @sarunas-merkliopas, @sbgraves237, Scott Ritchie, @scottko, @scottl, Sean Anderson, Sean Carmody, Sean Wilkinson, @sebastian-c, Sebastien Vigneau, @shabbychef, Shannon Rush, Simon O'Hanlon, Simon Potter, @SplashDance, @ste-fan, Stefan Widgren, @stephens999, Steven Pav, @strongh, @stuttungur, @surmann, @swnydick, @taekyunk, Tal Galili, @talgalili, @tdenes, @Thomas, @thomasherbig, @thomaszumbrunn, Tim Cole, @tjmahr, Tom Buckley, Tom Crockett, @ttriche, @twjacobs, @tyhenkaline, @tylerritchie, @ulrichatz, @varun729, @victorkryukov, @vijaybarve, @vzemlys, @wchi144, @wibeasley, @WilCrofter, William Doane, Winston Chang, @wmc3, @wordnerd, Yoni Ben-Meshulam, @zackham, @zerokarmaleft, Zhongpeng Lin.

1.7 Conventions

Throughout this book I use `f()` to refer to functions, `g` to refer to variables and function parameters, and `h/` to paths.

Larger code blocks intermingle input and output. Output is commented so that if you have an electronic version of the book, e.g., `http://adv-r.had.co.nz`, you can easily copy and paste examples into R. Output comments look like `#>` to distinguish them from regular comments.

1.8 Colophon

This book was written in Rmarkdown (`http://rmarkdown.rstudio.com/`) inside Rstudio (`http://www.rstudio.com/ide/`). knitr (`http://yihui.name/knitr/`) and pandoc (`http://johnmacfarlane.net/pandoc/`) converted the raw Rmarkdown to html and pdf. The website (`http://adv-r.had.co.nz`) was made with jekyll (`http://jekyllrb.com/`), styled with bootstrap (`http://getbootstrap.com/`), and automatically

published to Amazon's S3 (`http://aws.amazon.com/s3/`) by travis-ci (`https://travis-ci.org/`). The complete source is available from github (`https://github.com/hadley/adv-r`).

Code is set in inconsolata (`http://levien.com/type/myfonts/inconsolata.html`).

Part I

Foundations

2

Data structures

This chapter summarises the most important data structures in base R. You've probably used many (if not all) of them before, but you may not have thought deeply about how they are interrelated. In this brief overview, I won't discuss individual types in depth. Instead, I'll show you how they fit together as a whole. If you need more details, you can find them in R's documentation.

R's base data structures can be organised by their dimensionality (1d, 2d, or nd) and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types). This gives rise to the five data types most often used in data analysis:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Almost all other objects are built upon these foundations. In Chapter 7 you'll see how more complicated objects are built of these simple pieces. Note that R has no 0-dimensional, or scalar types. Individual numbers or strings, which you might think would be scalars, are actually vectors of length one.

Given an object, the best way to understand what data structures it's composed of is to use `str()`. `str()` is short for structure and it gives a compact, human readable description of any R data structure.

Quiz

Take this short quiz to determine if you need to read this chapter. If the answers quickly come to mind, you can comfortably skip this chapter. You can check your answers in Section 2.5.

1. What are the three properties of a vector, other than its contents?
2. What are the four common types of atomic vectors? What are the two rare types?
3. What are attributes? How do you get them and set them?
4. How is a list different from an atomic vector? How is a matrix different from a data frame?
5. Can you have a list that is a matrix? Can a data frame have a column that is a matrix?

Outline

- Section 2.1 introduces you to atomic vectors and lists, R's 1d data structures.
- Section 2.2 takes a small detour to discuss attributes, R's flexible metadata specification. Here you'll learn about factors, an important data structure created by setting attributes of an atomic vector.
- Section 2.3 introduces matrices and arrays, data structures for storing 2d and higher dimensional data.
- Section 2.4 teaches you about the data frame, the most important data structure for storing data in R. Data frames combine the behaviour of lists and matrices to make a structure ideally suited for the needs of statistical data.

2.1 Vectors

The basic data structure in R is the vector. Vectors come in two flavours: atomic vectors and lists. They have three common properties:

- Type, `typeof()`, what it is.
- Length, `length()`, how many elements it contains.
- Attributes, `attributes()`, additional arbitrary metadata.

They differ in the types of their elements: all elements of an atomic vector must be the same type, whereas the elements of a list can have different types.

NB: `is.vector()` does not test if an object is a vector. Instead it returns `TRUE` only if the object is a vector with no attributes apart from names. Use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

2.1.1 Atomic vectors

There are four common types of atomic vectors that I'll discuss in detail: logical, integer, double (often called numeric), and character. There are two rare types that I will not discuss further: complex and raw.

Atomic vectors are usually created with `c()`, short for combine:

```
dbl_var <- c(1, 2.5, 4.5)
# With the L suffix, you get an integer rather than a double
int_var <- c(1L, 6L, 10L)
# Use TRUE and FALSE (or T and F) to create logical vectors
log_var <- c(TRUE, FALSE, T, F)
chr_var <- c("these are", "some strings")
```

Atomic vectors are always flat, even if you nest `c()`'s:

```
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
# the same as
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

Missing values are specified with `NA`, which is a logical vector of length 1. `NA` will always be coerced to the correct type if used inside `c()`, or you can create `NA`s of a specific type with `NA_real_` (a double vector), `NA_integer_` and `NA_character_`.

2.1.1.1 Types and tests

Given a vector, you can determine its type with `typeof()`, or check if it's a specific type with an "is" function: `is.character()`, `is.double()`, `is.integer()`, `is.logical()`, or, more generally, `is.atomic()`.

```
int_var <- c(1L, 6L, 10L)
typeof(int_var)
#> [1] "integer"
```

```
is.integer(int_var)
#> [1] TRUE
is.atomic(int_var)
#> [1] TRUE

dbl_var <- c(1, 2.5, 4.5)
typeof(dbl_var)
#> [1] "double"
is.double(dbl_var)
#> [1] TRUE
is.atomic(dbl_var)
#> [1] TRUE
```

NB: `is.numeric()` is a general test for the "numberliness" of a vector and returns `TRUE` for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

```
is.numeric(int_var)
#> [1] TRUE
is.numeric(dbl_var)
#> [1] TRUE
```

2.1.1.2 Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be **coerced** to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

For example, combining a character and an integer yields a character:

```
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

When a logical vector is coerced to an integer or double, `TRUE` becomes 1 and `FALSE` becomes 0. This is very useful in conjunction with `sum()` and `mean()`

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1
```

```
# Total number of TRUEs
sum(x)
#> [1] 1

# Proportion that are TRUE
mean(x)
#> [1] 0.333
```

Coercion often happens automatically. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any`, etc) will coerce to a logical. You will usually get a warning message if the coercion might lose information. If confusion is likely, explicitly coerce with `as.character()`, `as.double()`, `as.integer()`, or `as.logical()`.

2.1.2 Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Lists are sometimes called **recursive** vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

`c()` will combine several lists into one. If given a combination of atomic vectors and lists, `c()` will coerce the vectors to lists before combining them. Compare the results of `list()` and `c()`:

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(y)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

The `typeof()` a list is `list`. You can test for a list with `is.list()` and coerce to a list with `as.list()`. You can turn a list into an atomic vector with `unlist()`. If the elements of a list have different types, `unlist()` uses the same coercion rules as `c()`.

Lists are used to build up many of the more complicated data structures in R. For example, both data frames (described in Section 2.4) and linear models objects (as produced by `lm()`) are lists:

```
is.list(mtcars)
#> [1] TRUE

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

2.1.3 Exercises

1. What are the six types of atomic vector? How does a list differ from an atomic vector?
2. What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?

3. Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

   ```
   c(1, FALSE)
   c("a", 1)
   c(list(1), "a")
   c(TRUE, 1L)
   ```

4. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?
5. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?
6. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

2.2 Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be thought of as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
y <- 1:10
attr(y, "my_attribute") <- "This is a vector"
attr(y, "my_attribute")
#> [1] "This is a vector"
str(attributes(y))
#> List of 1
#>  $ my_attribute: chr "This is a vector"
```

The `structure()` function returns a new object with modified attributes:

```
structure(1:10, my_attribute = "This is a vector")
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"my_attribute")
#> [1] "This is a vector"
```

By default, most attributes are lost when modifying a vector:

```
attributes(y[1])
#> NULL
attributes(sum(y))
#> NULL
```

The only attributes not lost are the three most important:

- Names, a character vector giving each element a name, described in Section 2.2.0.1.
- Dimensions, used to turn vectors into matrices and arrays, described in Section 2.3.
- Class, used to implement the S3 object system, described in Section 7.2.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `dim(x)`, and `class(x)`, not `attr(x, "names")`, `attr(x, "dim")`, and `attr(x, "class")`.

2.2.0.1 Names

You can name a vector in three ways:

- When creating it: `x <- c(a = 1, b = 2, c = 3)`.
- By modifying an existing vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`.
- By creating a modified copy of a vector: `x <- setNames(1:3, c("a", "b", "c"))`.

Names don't have to be unique. However, character subsetting, described in Section 3.4.1, is the most important reason to use names and it is most useful when the names are unique.

Not all elements of a vector need to have a name. If some names are missing, `names()` will return an empty string for those elements. If all names are missing, `names()` will return `NULL`.

```
y <- c(a = 1, 2, 3)
names(y)
#> [1] "a" ""  ""

z <- c(1, 2, 3)
names(z)
#> NULL
```

You can create a new vector without names using `unname(x)`, or remove names in place with `names(x) <- NULL`.

2.2.1 Factors

One important use of attributes is to define factors. A factor is a vector that can contain only predefined values, and is used to store categorical data. Factors are built on top of integer vectors using two attributes: the `class()`, "factor", which makes them behave differently from regular integer vectors, and the `levels()`, which defines the set of allowed values.

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
class(x)
#> [1] "factor"
levels(x)
#> [1] "a" "b"

# You can't use values that are not in the levels
x[2] <- "c"
#> Warning in `[<-.factor`(`*tmp*`, 2, value = "c"): invalid
#> factor level, NA generated
x
#> [1] a    <NA> b    a
#> Levels: a b

# NB: you can't combine factors
c(factor("a"), factor("b"))
#> [1] 1 1
```

Factors are useful when you know the possible values a variable may take, even if you don't see all values in a given dataset. Using a factor

instead of a character vector makes it obvious when some groups contain no observations:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
#> sex_char
#> m
#> 3
table(sex_factor)
#> sex_factor
#> m f
#> 3 0
```

Sometimes when a data frame is read directly from a file, a column you'd thought would produce a numeric vector instead produces a factor. This is caused by a non-numeric value in the column, often a missing value encoded in a special way like `.` or `-`. To remedy the situation, coerce the vector from a factor to a character vector, and then from a character to a double vector. (Be sure to check for missing values after this process.) Of course, a much better plan is to discover what caused the problem in the first place and fix that; using the `na.strings` argument to `read.csv()` is often a good place to start.

```
# Reading in "text" instead of from a file here:
z <- read.csv(text = "value\n12\n1\n.\n9")
typeof(z$value)
#> [1] "integer"
as.double(z$value)
#> [1] 3 2 1 4
# Oops, that's not right: 3 2 1 4 are the levels of a factor,
# not the values we read in!
class(z$value)
#> [1] "factor"
# We can fix it now:
as.double(as.character(z$value))
#> Warning: NAs introduced by coercion
#> [1] 12  1 NA  9
# Or change how we read it in:
z <- read.csv(text = "value\n12\n1\n.\n9", na.strings=".")
typeof(z$value)
```

```
#> [1] "integer"
class(z$value)
#> [1] "integer"
z$value
#> [1] 12  1 NA  9
# Perfect! :)
```

Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data. A global option, `options(stringsAsFactors = FALSE)`, is available to control this behaviour, but I don't recommend using it. Changing a global option may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing), and global options make code harder to understand because they increase the number of lines you need to read to understand how a single line of code will behave.

While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, it's usually best to explicitly convert factors to character vectors if you need string-like behaviour. In early versions of R, there was a memory advantage to using factors instead of character vectors, but this is no longer the case.

2.2.2 Exercises

1. An early draft used this code to illustrate `structure()`:

   ```
   structure(1:5, comment = "my attribute")
   #> [1] 1 2 3 4 5
   ```

   But when you print that object you don't see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.)

2. What happens to a factor when you modify its levels?

   ```
   f1 <- factor(letters)
   levels(f1) <- rev(levels(f1))
   ```

3. What does this code do? How do `f2` and `f3` differ from `f1`?

   ```
   f2 <- rev(factor(letters))

   f3 <- factor(letters, levels = rev(letters))
   ```

2.3 Matrices and arrays

Adding a `dim()` attribute to an atomic vector allows it to behave like a multi-dimensional **array**. A special case of the array is the **matrix**, which has two dimensions. Matrices are used commonly as part of the mathematical machinery of statistics. Arrays are much rarer, but worth being aware of.

Matrices and arrays are created with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments to specify rows and columns
a <- matrix(1:6, ncol = 3, nrow = 2)
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))

# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
dim(c) <- c(2, 3)
c
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

`length()` and `names()` have high-dimensional generalisations:

- `length()` generalises to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.

- `names()` generalises to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list of character vectors, for arrays.

```
length(a)
#> [1] 6
nrow(a)
#> [1] 2
ncol(a)
#> [1] 3
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
a
#>   a b c
#> A 1 3 5
#> B 2 4 6

length(b)
#> [1] 12
dim(b)
#> [1] 2 3 2
dimnames(b) <- list(c("one", "two"), c("a", "b", "c"), c("A", "B"))
b
#> , , A
#>
#>     a b c
#> one 1 3 5
#> two 2 4 6
#>
#> , , B
#>
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

`c()` generalises to `cbind()` and `rbind()` for matrices, and to `abind()` (provided by the `abind` package) for arrays. You can transpose a matrix with `t()`; the generalised equivalent for arrays is `aperm()`.

You can test if an object is a matrix or array using `is.matrix()` and `is.array()`, or by looking at the length of the `dim()`. `as.matrix()` and `as.array()` make it easy to turn an existing vector into a matrix or array.

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column, or arrays with a single

dimension. They may print similarly, but will behave differently. The differences aren't too important, but it's useful to know they exist in case you get strange output from a function (`tapply()` is a frequent offender). As always, use `str()` to reveal the differences.

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

While atomic vectors are most commonly turned into matrices, the dimension attribute can also be set on lists to make list-matrices or list-arrays:

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
l
#>      [,1]      [,2]
#> [1,] Integer,3 TRUE
#> [2,] "a"       1
```

These are relatively esoteric data structures, but can be useful if you want to arrange objects into a grid-like structure. For example, if you're running models on a spatio-temporal grid, it might be natural to preserve the grid structure by storing the models in a 3d array.

2.3.1 Exercises

1. What does `dim()` return when applied to a vector?
2. If `is.matrix(x)` is `TRUE`, what will `is.array(x)` return?
3. How would you describe the following three objects? What makes them different to `1:5`?

   ```
   x1 <- array(1:5, c(1, 1, 5))
   x2 <- array(1:5, c(1, 5, 1))
   x3 <- array(1:5, c(5, 1, 1))
   ```

2.4 Data frames

A data frame is the most common way of storing data in R, and if used systematically (http://vita.had.co.nz/papers/tidy-data.pdf) makes data analysis easier. Under the hood, a data frame is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()`, and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`; `nrow()` gives the number of rows.

As described in Chapter 3, you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

2.4.1 Creation

You create a data frame using `data.frame()`, which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Beware `data.frame()`'s default behaviour which turns strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour:

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

2.4.2 Testing and coercion

Because a `data.frame` is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use `class()` or test explicitly with `is.data.frame()`:

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

2.4.3 Combining data frames

You can combine data frames using `cbind()` and `rbind()`:

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number

and names of columns must match. Use `plyr::rbind.fill()` to combine data frames that don't have the same columns.

It's a common mistake to try and create a data frame by `cbind()`ing vectors together. This doesn't work because `cbind()` will create a matrix unless one of the arguments is already a data frame. Instead use `data.frame()` directly:

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: Factor w/ 2 levels "1","2": 1 2
#>  $ b: Factor w/ 2 levels "a","b": 1 2
good <- data.frame(a = 1:2, b = c("a", "b"),
  stringsAsFactors = FALSE)
str(good)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: int  1 2
#>  $ b: chr  "a" "b"
```

The conversion rules for `cbind()` are complicated and best avoided by ensuring all inputs are of the same type.

2.4.4 Special columns

Since a data frame is a list of vectors, it is possible for a data frame to have a column that is a list:

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```

However, when a list is given to `data.frame()`, it tries to put each item of the list into its own column, so this fails:

```
data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error in data.frame(1:2, 1:3, 1:4, check.names = FALSE, stringsAsFactors = TRUE): arguments im
```

A workaround is to use `I()`, which causes `data.frame()` to treat the list as one unit:

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y:List of 3
#>   ..$ : int  1 2
#>   ..$ : int  1 2 3
#>   ..$ : int  1 2 3 4
#>   ..- attr(*, "class")= chr "AsIs"
dfl[2, "y"]
#> [[1]]
#> [1] 1 2 3
```

`I()` adds the `AsIs` class to its input, but this can usually be safely ignored.

Similarly, it's also possible to have a column of a data frame that's a matrix or array, as long as the number of rows matches the data frame:

```
dfm <- data.frame(x = 1:3, y = I(matrix(1:9, nrow = 3)))
str(dfm)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: 'AsIs' int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
dfm[2, "y"]
#>      [,1] [,2] [,3]
#> [1,]    2    5    8
```

Use list and array columns with caution: many functions that work with data frames assume that all columns are atomic vectors.

2.4.5 Exercises

1. What attributes does a data frame possess?
2. What does `as.matrix()` do when applied to a data frame with columns of different types?
3. Can you have a data frame with 0 rows? What about 0 columns?

2.5 Answers

1. The three properties of a vector are type, length, and attributes.
2. The four common types of atomic vector are logical, integer, double (sometimes called numeric), and character. The two rarer types are complex and raw.
3. Attributes allow you to associate arbitrary additional metadata to any object. You can get and set individual attributes with `attr(x, "y")` and `attr(x, "y") <- value`; or get and set all attributes at once with `attributes()`.
4. The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type. Similarly, every element of a matrix must be the same type; in a data frame, the different columns can have different types.
5. You can make "list-array" by assuming dimensions to a list. You can make a matrix a column of a data frame with `df$x <- matrix()`, or using `I()` when creating a new data frame `data.frame(x = I(matrix()))`.

3

Subsetting

R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match. Subsetting is hard to learn because you need to master a number of interrelated concepts:

- The three subsetting operators.
- The six types of subsetting.
- Important differences in behaviour for different objects (e.g., vectors, lists, factors, matrices, and data frames).
- The use of subsetting in conjunction with assignment.

This chapter helps you master subsetting by starting with the simplest type of subsetting: subsetting an atomic vector with `[`. It then gradually extends your knowledge, first to more complicated data types (like arrays and lists), and then to the other subsetting operators, `[[` and `$`. You'll then learn how subsetting and assignment can be combined to modify parts of an object, and, finally, you'll see a large number of useful applications.

Subsetting is a natural complement to `str()`. `str()` shows you the structure of any object, and subsetting allows you to pull out the pieces that you're interested in.

Quiz

Take this short quiz to determine if you need to read this chapter. If the answers quickly come to mind, you can comfortably skip this chapter. Check your answers in Section 3.5.

1. What is the result of subsetting a vector with positive integers, negative integers, a logical vector, or a character vector?

2. What's the difference between `[`, `[[`, and `$` when applied to a list?
3. When should you use `drop = FALSE`?
4. If `x` is a matrix, what does `x[] <- 0` do? How is it different to `x <- 0`?
5. How can you use a named vector to relabel categorical variables?

Outline

- Section 3.1 starts by teaching you about `[`. You'll start by learning the six types of data that you can use to subset atomic vectors. You'll then learn how those six data types act when used to subset lists, matrices, data frames, and S3 objects.
- Section 3.2 expands your knowledge of subsetting operators to include `[[` and `$`, focussing on the important principles of simplifying vs. preserving.
- In Section 3.3 you'll learn the art of subassignment, combining subsetting and assignment to modify parts of an object.
- Section 3.4 leads you through eight important, but not obvious, applications of subsetting to solve problems that you often encounter in a data analysis.

3.1 Data types

It's easiest to learn how subsetting works for atomic vectors, and then how it generalises to higher dimensions and other more complicated objects. We'll start with `[`, the most commonly used operator. Section 3.2 will cover `[[` and `$`, the two other main subsetting operators.

3.1.1 Atomic vectors

Let's explore the different types of subsetting with a simple vector, `x`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Note that the number after the decimal point gives the original position in the vector.

There are five things that you can use to subset a vector:

- **Positive integers** return elements at the specified positions:

```
x[c(3, 1)]
#> [1] 3.3 2.1
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4

# Duplicated indices yield duplicated values
x[c(1, 1)]
#> [1] 2.1 2.1

# Real numbers are silently truncated to integers
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

- **Negative integers** omit elements at the specified positions:

```
x[-c(3, 1)]
#> [1] 4.2 5.4
```

  You can't mix positive and negative integers in a single subset:

```
x[c(-1, 2)]
#> Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

- **Logical vectors** select elements where the corresponding logical value is `TRUE`. This is probably the most useful type of subsetting because you write the expression that creates the logical vector:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
x[x > 3]
#> [1] 4.2 3.3 5.4
```

  If the logical vector is shorter than the vector being subsetted, it will be *recycled* to be the same length.

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

A missing value in the index always yields a missing value in the output:

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2  NA
```

- **Nothing** returns the original vector. This is not useful for vectors but is very useful for matrices, data frames, and arrays. It can also be useful in conjunction with assignment.

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

- **Zero** returns a zero-length vector. This is not something you usually do on purpose, but it can be helpful for generating test data.

```
x[0]
#> numeric(0)
```

If the vector is named, you can also use:

- **Character vectors** to return elements with matching names.

```
(y <- setNames(x, letters[1:4]))
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Like integer indices, you can repeat indices
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# When subsetting with [ names are always matched exactly
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

3.1.2 Lists

Subsetting a list works in the same way as subsetting an atomic vector. Using `[` will always return a list; `[[` and `$`, as described below, let you pull out the components of the list.

3.1.3 Matrices and arrays

You can subset higher-dimensional structures in three ways:

- With multiple vectors.
- With a single vector.
- With a matrix.

The most common way of subsetting matrices (2d) and arrays (>2d) is a simple generalisation of 1d subsetting: you supply a 1d index for each dimension, separated by a comma. Blank subsetting is now useful because it lets you keep all rows or all columns.

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(T, F, T), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
a[0, -2]
#>      A C
```

By default, `[` will simplify the results to the lowest possible dimensionality. See Section 3.2.1 to learn how to avoid this.

Because matrices and arrays are implemented as vectors with special attributes, you can subset them with a single vector. In that case, they will behave like a vector. Arrays in R are stored in column-major order:

```
(vals <- outer(1:5, 1:5, FUN = "paste", sep = ","))
#>      [,1]  [,2]  [,3]  [,4]  [,5]
```

```
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

You can also subset higher-dimensional data structures with an integer matrix (or, if named, a character matrix). Each row in the matrix specifies the location of one value, where each column corresponds to a dimension in the array being subsetted. This means that you use a 2 column matrix to subset a matrix, a 3 column matrix to subset a 3d array, and so on. The result is a vector of values:

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

3.1.4 Data frames

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])

df[df$x == 2, ]
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c

# There are two ways to select columns from a data frame
```

```
# Like a list:
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Like a matrix
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c

# There's an important difference if you select a single
# column: matrix subsetting simplifies by default, list
# subsetting does not.
str(df["x"])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
str(df[, "x"])
#>  int [1:3] 1 2 3
```

3.1.5 S3 objects

S3 objects are made up of atomic vectors, arrays, and lists, so you can always pull apart an S3 object using the techniques described above and the knowledge you gain from `str()`.

3.1.6 S4 objects

There are also two additional subsetting operators that are needed for S4 objects: `@` (equivalent to `$`), and `slot()` (equivalent to `[[`). `@` is more restrictive than `$` in that it will return an error if the slot does not exist. These are described in more detail in Section 7.3.

3.1.7 Exercises

1. Fix each of the following common data frame subsetting errors:

```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

2. Why does `x <- 1:5; x[NA]` yield five missing values? (Hint: why is it different from `x[NA_real_]`?)
3. What does `upper.tri()` return? How does subsetting a matrix with it work? Do we need any additional subsetting rules to describe its behaviour?

   ```
   x <- outer(1:5, 1:5, FUN = "*")
   x[upper.tri(x)]
   ```

4. Why does `mtcars[1:20]` return a error? How does it differ from the similar `mtcars[1:20, ]`?
5. Implement your own function that extracts the diagonal entries from a matrix (it should behave like `diag(x)` where `x` is a matrix).
6. What does `df[is.na(df)] <- 0` do? How does it work?

3.2 Subsetting operators

There are two other subsetting operators: `[[` and `$`. `[[` is similar to `[`, except it can only return a single value and it allows you to pull pieces out of a list. `$` is a useful shorthand for `[[` combined with character subsetting.

You need `[[` when working with lists. This is because when `[` is applied to a list it always returns a list: it never gives you the contents of the list. To get the contents, you need `[[`:

> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6."
>
> — @RLangTip

Because it can return only a single value, you must use `[[` with either a single positive integer or a string:

```
a <- list(a = 1, b = 2)
a[[1]]
#> [1] 1
a[["a"]]
#> [1] 1

# If you do supply a vector it indexes recursively
b <- list(a = list(b = list(c = list(d = 1))))
b[[c("a", "b", "c", "d")]]
#> [1] 1
# Same as
b[["a"]][["b"]][["c"]][["d"]]
#> [1] 1
```

Because data frames are lists of columns, you can use `[[` to extract a column from data frames: `mtcars[[1]]`, `mtcars[["cyl"]]`.

S3 and S4 objects can override the standard behaviour of `[` and `[[` so they behave differently for different types of objects. The key difference is usually how you select between simplifying or preserving behaviours, and what the default is.

3.2.1 Simplifying vs. preserving subsetting

It's important to understand the distinction between simplifying and preserving subsetting. Simplifying subsets returns the simplest possible data structure that can represent the output, and is useful interactively because it usually gives you what you want. Preserving subsetting keeps the structure of the output the same as the input, and is generally better for programming because the result will always be the same type. Omitting `drop = FALSE` when subsetting matrices and data frames is one of the most common sources of programming errors. (It will work for your test cases, but then someone will pass in a single column data frame and it will fail in an unexpected and unclear way.)

Unfortunately, how you switch between simplifying and preserving differs for different data types, as summarised in the table below.

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`

	Simplifying	Preserving
Array	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = F]` **or** `x[, 1, drop = F]`
Data frame	`x[, 1]` **or** `x[[1]]`	`x[, 1, drop = F]` **or** `x[1]`

Preserving is the same for all data types: you get the same type of output as input. Simplifying behaviour varies slightly between different data types, as described below:

- **Atomic vector**: removes names.

```
x <- c(a = 1, b = 2)
x[1]
#> a
#> 1
x[[1]]
#> [1] 1
```

- **List**: return the object inside the list, not a single element list.

```
y <- list(a = 1, b = 2)
str(y[1])
#> List of 1
#>  $ a: num 1
str(y[[1]])
#>  num 1
```

- **Factor**: drops any unused levels.

```
z <- factor(c("a", "b"))
z[1]
#> [1] a
#> Levels: a b
z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

- **Matrix** or **array**: if any of the dimensions has length 1, drops that dimension.

```
a <- matrix(1:4, nrow = 2)
a[1, , drop = FALSE]
#>      [,1] [,2]
#> [1,]    1    3
```

```
a[1, ]
#> [1] 1 3
```

- **Data frame**: if output is a single column, returns a vector instead of a data frame.

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[1])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[[1]])
#>  int [1:2] 1 2
str(df[, "a", drop = FALSE])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[, "a"])
#>  int [1:2] 1 2
```

3.2.2 `$`

`$` is a shorthand operator, where `x$y` is equivalent to `x[["y", exact = FALSE]]`. It's often used to access variables in a data frame, as in `mtcars$cyl` or `diamonds$carat`.

One common mistake with `$` is to try and use it when you have the name of a column stored in a variable:

```
var <- "cyl"
# Doesn't work - mtcars$var translated to mtcars[["var"]]
mtcars$var
#> NULL

# Instead use [[
mtcars[[var]]
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8
#> [30] 6 8 4
```

There's one important difference between `$` and `[[`. `$` does partial matching:

```
x <- list(abc = 1)
x$a
```

```
#> [1] 1
x[["a"]]
#> NULL
```

If you want to avoid this behaviour you can set the global option `warnPartialMatchDollar` to `TRUE`. Use with caution: it may affect behaviour in other code you have loaded (e.g., from a package).

3.2.3 Missing/out of bounds indices

`[` and `[[` differ slightly in their behaviour when the index is out of bounds (OOB), for example, when you try to extract the fifth element of a length four vector, or subset a vector with `NA` or `NULL`:

```
x <- 1:4
str(x[5])
#>  int NA
str(x[NA_real_])
#>  int NA
str(x[NULL])
#>  int(0)
```

The following table summarises the results of subsetting atomic vectors and lists with `[` and `[[` and different types of OOB value.

Operator	Index	Atomic	List
`[`	OOB	`NA`	`list(NULL)`
`[`	`NA_real_`	`NA`	`list(NULL)`
`[`	`NULL`	`x[0]`	`list(NULL)`
`[[`	OOB	Error	Error
`[[`	`NA_real_`	Error	`NULL`
`[[`	`NULL`	Error	Error

If the input vector is named, then the names of OOB, missing, or `NULL` components will be `"<NA>"`.

3.2.4 Exercises

1. Given a linear model, e.g., `mod <- lm(mpg ~ wt, data = mtcars)`, extract the residual degrees of freedom. Extract the R squared from the model summary (`summary(mod)`)

3.3 Subsetting and assignment

All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5

# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Note that there's no checking for duplicate indices
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1

# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)
#> Error in x[c(1, NA)] <- c(1, 2): NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they're treated as false).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1

# This is mostly useful when conditionally modifying vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1]  0 10 NA
```

Subsetting with nothing can be useful in conjunction with assignment because it will preserve the original object class and structure. Compare the following two expressions. In the first, `mtcars` will remain as a data frame. In the second, `mtcars` will become a list.

```
mtcars[] <- lapply(mtcars, as.integer)
mtcars <- lapply(mtcars, as.integer)
```

With lists, you can use subsetting + assignment + `NULL` to remove components from a list. To add a literal `NULL` to a list, use `[` and `list(NULL)`:

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#>  $ a: num 1

y <- list(a = 1)
y["b"] <- list(NULL)
str(y)
#> List of 2
#>  $ a: num 1
#>  $ b: NULL
```

3.4 Applications

The basic principles described above give rise to a wide variety of useful applications. Some of the most important are described below. Many of these basic techniques are wrapped up into more concise functions (e.g., `subset()`, `merge()`, `plyr::arrange()`), but it is useful to understand how they are implemented with basic subsetting. This will allow you to adapt to new situations that are not dealt with by existing functions.

3.4.1 Lookup tables (character subsetting)

Character matching provides a powerful way to make lookup tables. Say you want to convert abbreviations:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
#>        m        f        u        f        f        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"
#>        m
#>   "Male"
unname(lookup[x])
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"
#> [7] "Male"

# Or with fewer output values
c(m = "Known", f = "Known", u = "Unknown")[x]
#>         m         f         u         f         f         m
#>   "Known"   "Known" "Unknown"   "Known"   "Known"   "Known"
#>         m
#>   "Known"
```

If you don't want names in the result, use `unname()` to remove them.

3.4.2 Matching and merging by hand (integer subsetting)

You may have a more complicated lookup table which has multiple columns of information. Suppose we have a vector of integer grades, and a table that describes their properties:

```
grades <- c(1, 2, 2, 3, 1)

info <- data.frame(
  grade = 3:1,
  desc = c("Excellent", "Good", "Poor"),
  fail = c(F, F, T)
)
```

We want to duplicate the info table so that we have a row for each value in `grades`. We can do this in two ways, either using `match()` and integer subsetting, or `rownames()` and character subsetting:

```
grades
#> [1] 1 2 2 3 1
```

```
# Using match
id <- match(grades, info$grade)
info[id, ]
#>     grade      desc  fail
#> 3       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 1       3 Excellent FALSE
#> 3.1     1      Poor  TRUE

# Using rownames
rownames(info) <- info$grade
info[as.character(grades), ]
#>     grade      desc  fail
#> 1       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 3       3 Excellent FALSE
#> 1.1     1      Poor  TRUE
```

If you have multiple columns to match on, you'll need to first collapse them to a single column (with `interaction()`, `paste()`, or `plyr::id()`). You can also use `merge()` or `plyr::join()`, which do the same thing for you — read the source code to see how.

3.4.3 Random samples/bootstrap (integer subsetting)

You can use integer indices to perform random sampling or bootstrapping of a vector or data frame. `sample()` generates a vector of indices, then subsetting to access the values:

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])

# Set seed for reproducibility
set.seed(10)

# Randomly reorder
df[sample(nrow(df)), ]
#>   x y z
#> 4 2 3 d
#> 2 1 5 b
#> 5 3 2 e
```

```
#> 3 2 4 c
#> 1 1 6 a
#> 6 3 1 f
# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 2 1 5 b
#> 6 3 1 f
#> 3 2 4 c
# Select 6 bootstrap replicates
df[sample(nrow(df), 6, rep = T), ]
#>     x y z
#> 3   2 4 c
#> 4   2 3 d
#> 4.1 2 3 d
#> 1   1 6 a
#> 4.2 2 3 d
#> 3.1 2 4 c
```

The arguments of `sample()` control the number of samples to extract, and whether sampling is performed with or without replacement.

3.4.4 Ordering (integer subsetting)

`order()` takes a vector as input and returns an integer vector describing how the subsetted vector should be ordered:

```
x <- c("b", "c", "a")
order(x)
#> [1] 3 1 2
x[order(x)]
#> [1] "a" "b" "c"
```

To break ties, you can supply additional variables to `order()`, and you can change from ascending to descending order using `decreasing = TRUE`. By default, any missing values will be put at the end of the vector; however, you can remove them with `na.last = NA` or put at the front with `na.last = FALSE`.

For two or more dimensions, `order()` and integer subsetting makes it easy to order either the rows or columns of an object:

```
# Randomly reorder df
df2 <- df[sample(nrow(df)), 3:1]
df2
#>   z y x
#> 3 c 4 2
#> 1 a 6 1
#> 2 b 5 1
#> 4 d 3 2
#> 6 f 1 3
#> 5 e 2 3

df2[order(df2$x), ]
#>   z y x
#> 1 a 6 1
#> 2 b 5 1
#> 3 c 4 2
#> 4 d 3 2
#> 6 f 1 3
#> 5 e 2 3
df2[, order(names(df2))]
#>   x y z
#> 3 2 4 c
#> 1 1 6 a
#> 2 1 5 b
#> 4 2 3 d
#> 6 3 1 f
#> 5 3 2 e
```

More concise, but less flexible, functions are available for sorting vectors, `sort()`, and data frames, `plyr::arrange()`.

3.4.5 Expanding aggregated counts (integer subsetting)

Sometimes you get a data frame where identical rows have been collapsed into one and a count column has been added. `rep()` and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index:

```
df <- data.frame(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
rep(1:nrow(df), df$n)
#> [1] 1 1 1 2 2 2 2 2 3
df[rep(1:nrow(df), df$n), ]
```

```
#>     x  y n
#> 1   2  9 3
#> 1.1 2  9 3
#> 1.2 2  9 3
#> 2   4 11 5
#> 2.1 4 11 5
#> 2.2 4 11 5
#> 2.3 4 11 5
#> 2.4 4 11 5
#> 3   1  6 1
```

3.4.6 Removing columns from data frames (character subsetting)

There are two ways to remove columns from a data frame. You can set individual columns to NULL:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df$z <- NULL
```

Or you can subset to return only the columns you want:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[c("x", "y")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

If you know the columns you don't want, use set operations to work out which colums to keep:

```
df[setdiff(names(df), "z")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

3.4.7 Selecting rows based on a condition (logical subsetting)

Because it allows you to easily combine conditions from multiple columns, logical subsetting is probably the most commonly used technique for extracting rows out of a data frame.

```
mtcars[mtcars$gear == 5, ]
#>     mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.17 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.77 15.5  0  1    5    6
#> 31 15.0   8 301.0 335 3.54 3.57 14.6  0  1    5    8
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
#>     mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
```

Remember to use the vector boolean operators `&` and `|`, not the short-circuiting scalar operators `&&` and `||` which are more useful inside if statements. Don't forget De Morgan's laws (`http://en.wikipedia.org/wiki/De_Morgan's_laws`), which can be useful to simplify negations:

- `!(X & Y)` is the same as `!X | !Y`
- `!(X | Y)` is the same as `!X & !Y`

For example, `!(X & !(Y | Z))` simplifies to `!X | !!(Y|Z)`, and then to `!X | Y | Z`.

`subset()` is a specialised shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame. You'll learn how it works in Chapter 13.

```
subset(mtcars, gear == 5)
#>     mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.17 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.77 15.5  0  1    5    6
#> 31 15.0   8 301.0 335 3.54 3.57 14.6  0  1    5    8
subset(mtcars, gear == 5 & cyl == 4)
```

```
#>     mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
```

3.4.8 Boolean algebra vs. sets (logical & integer subsetting)

It's useful to be aware of the natural equivalence between set operations (integer subsetting) and boolean algebra (logical subsetting). Using set operations is more effective when:

- You want to find the first (or last) `TRUE`.
- You have very few `TRUE`s and very many `FALSE`s; a set representation may be faster and require less storage.

`which()` allows you to convert a boolean representation to an integer representation. There's no reverse operation in base R but we can easily create one:

```
x <- sample(10) < 4
which(x)
#> [1]  3  7 10

unwhich <- function(x, n) {
  out <- rep_len(FALSE, n)
  out[x] <- TRUE
  out
}
unwhich(which(x), 10)
#>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
#> [10]  TRUE
```

Let's create two logical vectors and their integer equivalents and then explore the relationship between boolean and set operations.

```
(x1 <- 1:10 %% 2 == 0)
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
#> [10]  TRUE
(x2 <- which(x1))
#> [1]  2  4  6  8 10
```

```
(y1 <- 1:10 %% 5 == 0)
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
#> [10]  TRUE
(y2 <- which(y1))
#> [1]  5 10

# X & Y <-> intersect(x, y)
x1 & y1
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [10]  TRUE
intersect(x2, y2)
#> [1] 10

# X | Y <-> union(x, y)
x1 | y1
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
#> [10]  TRUE
union(x2, y2)
#> [1]  2  4  6  8 10  5

# X & !Y <-> setdiff(x, y)
x1 & !y1
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
#> [10] FALSE
setdiff(x2, y2)
#> [1] 2 4 6 8

# xor(X, Y) <-> setdiff(union(x, y), intersect(x, y))
xor(x1, y1)
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
#> [10] FALSE
setdiff(union(x2, y2), intersect(x2, y2))
#> [1] 2 4 6 8 5
```

When first learning subsetting, a common mistake is to use `x[which(y)]` instead of `x[y]`. Here the `which()` achieves nothing: it switches from logical to integer subsetting but the result will be exactly the same. Also beware that `x[-which(y)]` is **not** equivalent to `x[!y]`: if `y` is all FALSE, `which(y)` will be `integer(0)` and `-integer(0)` is still `integer(0)`, so you'll get no values, instead of all values. In general, avoid switching from logical to integer subsetting unless you want, for example, the first or last `TRUE` value.

3.4.9 Exercises

1. How would you randomly permute the columns of a data frame? (This is an important technique in random forests.) Can you simultaneously permute the rows and columns in one step?
2. How would you select a random sample of `m` rows from a data frame? What if the sample had to be contiguous (i.e., with an initial row, a final row, and every row in between)?
3. How could you put the columns in a data frame in alphabetical order?

3.5 Answers

1. Positive integers select elements at specific positions, negative integers drop elements; logical vectors keep elements at positions corresponding to `TRUE`; character vectors select elements with matching names.
2. `[` selects sub-lists. It always returns a list; if you use it with a single positive integer, it returns a list of length one. `[[` selects an element within a list. `$` is a convenient shorthand: `x$y` is equivalent to `x[["y"]]`.
3. Use `drop = FALSE` if you are subsetting a matrix, array, or data frame and you want to preserve the original dimensions. You should almost always use it when subsetting inside a function.
4. If `x` is a matrix, `x[] <- 0` will replace every element with 0, keeping the same number of rows and columns. `x <- 0` completely replaces the matrix with the value 0.
5. A named character vector can act as a simple lookup table: `c(x = 1, y = 2, z = 3)[c("y", "z", "x")]`

4

Vocabulary

An important part of being fluent in R is having a good working vocabulary. Below, I have listed the functions that I believe constitute such a vocabulary. You don't need to be intimately familiar with the details of every function, but you should at least be aware that they all exist. If there are functions in this list that you've never heard of, I strongly recommend that you read their documentation.

I came up with this list by looking through all the functions in the base, stats, and utils packages, and extracting those that I think are most useful. The list also includes a few pointers to particularly important functions in other packages, and some of the more important `options()`.

4.1 The basics

```
# The first functions to learn
?
str

# Important operators and assignment
%in%, match
=, <-, <<-
$, [, [[, head, tail, subset
with
assign, get

# Comparison
all.equal, identical
!=, ==, >, >=, <, <=
is.na, complete.cases
is.finite
```

```
# Basic math
*, +, -, /, ^, %%, %/%
abs, sign
acos, asin, atan, atan2
sin, cos, tan
ceiling, floor, round, trunc, signif
exp, log, log10, log2, sqrt

max, min, prod, sum
cummax, cummin, cumprod, cumsum, diff
pmax, pmin
range
mean, median, cor, sd, var
rle

# Functions to do with functions
function
missing
on.exit
return, invisible

# Logical & sets
&, |, !, xor
all, any
intersect, union, setdiff, setequal
which

# Vectors and matrices
c, matrix
# automatic coercion rules character > numeric > logical
length, dim, ncol, nrow
cbind, rbind
names, colnames, rownames
t
diag
sweep
as.matrix, data.matrix

# Making vectors
c
rep, rep_len
seq, seq_len, seq_along
```

```
rev
sample
choose, factorial, combn
(is/as).(character/numeric/logical/...)

# Lists & data.frames
list, unlist
data.frame, as.data.frame
split
expand.grid

# Control flow
if, &&, || (short circuiting)
for, while
next, break
switch
ifelse

# Apply & friends
lapply, sapply, vapply
apply
tapply
replicate
```

4.2 Common data structures

```
# Date time
ISOdate, ISOdatetime, strftime, strptime, date
difftime
julian, months, quarters, weekdays
library(lubridate)

# Character manipulation
grep, agrep
gsub
strsplit
chartr
nchar
tolower, toupper
```

```
substr
paste
library(stringr)

# Factors
factor, levels, nlevels
reorder, relevel
cut, findInterval
interaction
options(stringsAsFactors = FALSE)

# Array manipulation
array
dim
dimnames
aperm
library(abind)
```

4.3 Statistics

```
# Ordering and tabulating
duplicated, unique
merge
order, rank, quantile
sort
table, ftable

# Linear models
fitted, predict, resid, rstandard
lm, glm
hat, influence.measures
logLik, df, deviance
formula, ~, I
anova, coef, confint, vcov
contrasts

# Miscellaneous tests
apropos("\\.test$")
```

```
# Random variables
(q, p, d, r) * (beta, binom, cauchy, chisq, exp, f, gamma, geom,
  hyper, lnorm, logis, multinom, nbinom, norm, pois, signrank, t,
  unif, weibull, wilcox, birthday, tukey)

# Matrix algebra
crossprod, tcrossprod
eigen, qr, svd
%*%, %o%, outer
rcond
solve
```

4.4 Working with R

```
# Workspace
ls, exists, rm
getwd, setwd
q
source
install.packages, library, require

# Help
help, ?
help.search
apropos
RSiteSearch
citation
demo
example
vignette

# Debugging
traceback
browser
recover
options(error = )
stop, warning, message
tryCatch, try
```

4.5 I/O

```
# Output
print, cat
message, warning
dput
format
sink, capture.output

# Reading and writing data
data
count.fields
read.csv, write.csv
read.delim, write.delim
read.fwf
readLines, writeLines
readRDS, saveRDS
load, save
library(foreign)

# Files and directories
dir
basename, dirname, tools::file_ext
file.path
path.expand, normalizePath
file.choose
file.copy, file.create, file.remove, file.rename, dir.create
file.exists, file.info
tempdir, tempfile
download.file, library(downloader)
```

5

Style guide

Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations. The following guide describes the style that I use (in this book and elsewhere). It is based on Google's R style guide (`http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`), with a few tweaks. You don't have to use my style, but you really should use a consistent style.

Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you'll need to sacrifice some preferred aspects of your style.

The formatR package, by Yihui Xie, makes it easier to clean up poorly formatted code. It can't do everything, but it can quickly get your code from terrible to pretty good. Make sure to read the introduction (`http://yihui.name/formatR/`) before using it.

5.1 Notation and naming

5.1.1 File names

File names should be meaningful and end in `.R`.

```
# Good
fit-models.R
utility-functions.R

# Bad
```

```
foo.r
stuff.r
```

If files need to be run in sequence, prefix them with numbers:

```
0-download.R
1-parse.R
2-explore.R
```

5.1.2 Object names

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> — Phil Karlton

Variable and function names should be lowercase. Use an underscore (_) to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

Where possible, avoid using names of existing functions and variables. This will cause confusion for the readers of your code.

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

5.2 Syntax

5.2.1 Spacing

Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.). The same rule applies when using `=` in function calls. Always put a space after a comma, and never before (just like in regular English).

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

There's a small exception to this rule: `:`, `::` and `:::` don't need spaces around them.

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

Place a space before left parentheses, except in a function call.

```
# Good
if (debug) do(x)
plot(x, y)

# Bad
if(debug)do(x)
plot (x, y)
```

Extra spacing (i.e., more than one space in a row) is ok if it improves alignment of equal signs or assignments (`<-`).

```
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

Do not place spaces around code in parentheses or square brackets (unless there's a comma, in which case see above).

```
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

5.2.2 Curly braces

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it's followed by `else`.

Always indent the code inside curly braces.

```
# Good

if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad

if (y < 0 && debug)
message("Y is negative")
```

```
if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

It's ok to leave very short statements on the same line:

```
if (y < 0 && debug) message("Y is negative")
```

5.2.3 Line length

Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font. If you find yourself running out of room, this is a good indication that you should encapsulate some of the work in a separate function.

5.2.4 Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces.

The only exception is if a function definition runs over multiple lines. In that case, indent the second line to where the definition starts:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

5.2.5 Assignment

Use `<-`, not `=`, for assignment.

```
# Good
x <- 5
# Bad
x = 5
```

5.3 Organisation

5.3.1 Commenting guidelines

Comment your code. Each line of a comment should begin with the comment symbol and a single space: `#`. Comments should explain the why, not the what.

Use commented lines of `-` and `=` to break up your file into easily readable chunks.

```
# Load data --------------------------

# Plot data --------------------------
```

6

Functions

Functions are a fundamental building block of R: to master many of the more advanced techniques in this book, you need a solid foundation in how functions work. You've probably already created many R functions, and you're familiar with the basics of how they work. The focus of this chapter is to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how they work. You'll see some interesting tricks and techniques in this chapter, but most of what you'll learn will be more important as the building blocks for more advanced techniques.

The most important thing to understand about R is that functions are objects in their own right. You can work with them exactly the same way you work with any other type of object. This theme will be explored in depth in Chapter 10.

Quiz

Answer the following questions to see if you can safely skip this chapter. You can find the answers at the end of the chapter in Section 6.7.

1. What are the three components of a function?
2. What does the following code return?

```
x <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

3. How would you more typically write this code?

```
`+`(1, `*`(2, 3))
```

4. How could you make this call easier to read?

   ```
   mean(, TRUE, x = c(1:10, NA))
   ```

5. Does the following function throw an error when called? Why/why not?

   ```
   f2 <- function(a, b) {
     a * 10
   }
   f2(10, stop("This is an error!"))
   ```

6. What is an infix function? How do you write it? What's a replacement function? How do you write it?
7. What function do you use to ensure that a cleanup action occurs regardless of how a function terminates?

Outline

- Section 6.1 describes the three main components of a function.
- Section 6.2 teaches you how R finds values from names, the process of lexical scoping.
- Section 6.3 shows you that everything that happens in R is a result of a function call, even if it doesn't look like it.
- Section 6.4 discusses the three ways of supplying arguments to a function, how to call a function given a list of arguments, and the impact of lazy evaluation.
- Section 6.5 describes two special types of function: infix and replacement functions.
- Section 6.6 discusses how and when functions return values, and how you can ensure that a function does something before it exits.

Prerequisites

The only package you'll need is `pryr`, which is used to explore what happens when modifying vectors in place. Install it with `install.packages("pryr")`.

6.1 Function components

All R functions have three parts:

- the `body()`, the code inside the function.
- the `formals()`, the list of arguments which controls how you can call the function.
- the `environment()`, the "map" of the location of the function's variables.

When you print a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.

```
f <- function(x) x^2
f
#> function(x) x^2

formals(f)
#> $x
body(f)
#> x^2
environment(f)
#> <environment: R_GlobalEnv>
```

The assignment forms of `body()`, `formals()`, and `environment()` can also be used to modify functions.

Like all objects in R, functions can also possess any number of additional `attributes()`. One attribute used by base R is "srcref", short for source reference, which points to the source code used to create the function. Unlike `body()`, this contains code comments and other formatting. You can also add attributes to a function. For example, you can set the `class()` and add a custom `print()` method.

6.1.1 Primitive functions

There is one exception to the rule that functions have three components. Primitive functions, like `sum()`, call C code directly with `.Primitive()`

and contain no R code. Therefore their `formals()`, `body()`, and `environment()` are all `NULL`:

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Primitive functions are only found in the `base` package, and since they operate at a low level, they can be more efficient (primitive replacement functions don't have to make copies), and can have different rules for argument matching (e.g., `switch` and `call`). This, however, comes at a cost of behaving differently from all other functions in R. Hence the R core team generally avoids creating them unless there is no other option.

6.1.2 Exercises

1. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?
2. This code makes a list of all functions in the base package.

   ```
   objs <- mget(ls("package:base"), inherits = TRUE)
   funs <- Filter(is.function, objs)
   ```

   Use it to answer the following questions:

   a. Which base function has the most arguments?
   b. How many base functions have no arguments? What's special about those functions?
   c. How could you adapt the code to find all primitive functions?
3. What are the three important components of a function?
4. When does printing a function not show what environment it was created in?

6.2 Lexical scoping

Scoping is the set of rules that govern how R looks up the value of a symbol. In the example below, scoping is the set of rules that R applies to go from the symbol `x` to its value `10`:

```
x <- 10
x
#> [1] 10
```

Understanding scoping allows you to:

- build tools by composing functions, as described in Chapter 10.
- overrule the usual evaluation rules and do non-standard evaluation, as described in Chapter 13.

R has two types of scoping: **lexical scoping**, implemented automatically at the language level, and **dynamic scoping**, used in select functions to save typing during interactive analysis. We discuss lexical scoping here because it is intimately tied to function creation. Dynamic scoping is described in more detail in Section 13.3.

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called. With lexical scoping, you don't need to know how the function is called to figure out where the value of a variable will be looked up. You just need to look at the function's definition.

The "lexical" in lexical scoping doesn't correspond to the usual English definition ("of or relating to words or the vocabulary of a language as distinguished from its grammar and construction") but comes from the computer science term "lexing", which is part of the process that converts code represented as text to meaningful pieces that the programming language understands.

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables

- a fresh start
- dynamic lookup

You probably know many of these principles already, although you might not have thought about them explicitly. Test your knowledge by mentally running through the code in each block before looking at the answers.

6.2.1 Name masking

The following example illustrates the most basic principle of lexical scoping, and you should have no problem predicting the output.

```
f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
f()
rm(f)
```

If a name isn't defined inside a function, R will look one level up.

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()
rm(x, g)
```

The same rules apply if a function is defined inside another function: look inside the current function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages. Run the following code in your head, then confirm the output by running the R code.

```
x <- 1
h <- function() {
  y <- 2
```

```
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
h()
rm(x, h)
```

The same rules apply to closures, functions created by other functions. Closures will be described in more detail in Chapter 10; here we'll just look at how they interact with scoping. The following function, `j()`, returns a function. What do you think this function will return when we call it?

```
j <- function(x) {
  y <- 2
  function() {
    c(x, y)
  }
}
k <- j(1)
k()
rm(j, k)
```

This seems a little magical (how does R know what the value of `y` is after the function has been called). It works because `k` preserves the environment in which it was defined and because the environment includes the value of `y`. Chapter 8 gives some pointers on how you can dive in and figure out what values are stored in the environment associated with each function.

6.2.2 Functions vs. variables

The same principles apply regardless of the type of associated value — finding functions works exactly the same way as finding variables:

```
l <- function(x) x + 1
m <- function() {
  l <- function(x) x * 2
  l(10)
```

```
}
m()
#> [1] 20
rm(l, m)
```

For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g., `f(3)`), R will ignore objects that are not functions while it is searching. In the following example `n` takes on a different value depending on whether R is looking for a function or a variable.

```
n <- function(x) x / 2
o <- function() {
  n <- 10
  n(n)
}
o()
#> [1] 5
rm(n, o)
```

However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.

6.2.3 A fresh start

What happens to the values in between invocations of a function? What will happen the first time you run this function? What will happen the second time? (If you haven't seen `exists()` before: it returns `TRUE` if there's a variable of that name, otherwise it returns `FALSE`.)

```
j <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  print(a)
}
j()
rm(j)
```

You might be surprised that it returns the same value, 1, every time. This is because every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent. (We'll see some ways to get around this in Section 10.3.2.)

6.2.4 Dynamic lookup

Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created. This means that the output of a function can be different depending on objects outside its environment:

```
f <- function() x
x <- 15
f()
#> [1] 15

x <- 20
f()
#> [1] 20
```

You generally want to avoid this behaviour because it means the function is no longer self-contained. This is a common error — if you make a spelling mistake in your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.

One way to detect this problem is the `findGlobals()` function from `codetools`. This function lists all the external dependencies of a function:

```
f <- function() x + 1
codetools::findGlobals(f)
#> [1] "+" "x"
```

Another way to try and solve the problem would be to manually change the environment of the function to the `emptyenv()`, an environment which contains absolutely nothing:

```
environment(f) <- emptyenv()
f()
#> Error in f(): could not find function "+"
```

This doesn't work because R relies on lexical scoping to find *everything*, even the + operator. It's never possible to make a function completely self-contained because you must always rely on functions defined in base R or other packages.

You can use this same idea to do other things that are extremely ill-advised. For example, since all of the standard operators in R are functions, you can override them with your own alternatives. If you ever are feeling particularly evil, run the following code while your friend is away from their computer:

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 4 3 3 3 4 3 3 3 3 3 3 4 3 3 3 3 4 3 3 3 3 3 3 3 4 3 3
#> [30] 3 4 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 3
rm("(")
```

This will introduce a particularly pernicious bug: 10% of the time, 1 will be added to any numeric calculation inside parentheses. This is another good reason to regularly restart with a clean R session!

6.2.5 Exercises

1. What does the following code return? Why? What does each of the three `c`'s mean?

   ```
   c <- 10
   c(c = c)
   ```

2. What are the four principles that govern how R looks for values?

3. What does the following function return? Make a prediction before running the code yourself.

   ```
   f <- function(x) {
     f <- function(x) {
       f <- function(x) {
   ```

```
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

6.3 Every operation is a function call

"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 previous example of redefining `(` works because every operation in R is a function call, whether or not it looks like one. This includes infix operators like `+`, control flow operators like `for`, `if`, and `while`, subsetting operators like `[]` and `$`, and even the curly brace `{`. This means that each pair of statements in the following example is exactly equivalent. Note that `` ` ``, the backtick, lets you refer to functions or variables that have otherwise reserved or illegal names:

```
x <- 10; y <- 5
x + y
#> [1] 15
`+`(x, y)
#> [1] 15

for (i in 1:2) print(i)
#> [1] 1
#> [1] 2
`for`(i, 1:2, print(i))
#> [1] 1
#> [1] 2

if (i == 1) print("yes!") else print("no.")
```

```
#> [1] "no."
`if`(i == 1, print("yes!"), print("no."))
#> [1] "no."

x[3]
#> [1] NA
`[`(x, 3)
#> [1] NA

{ print(1); print(2); print(3) }
#> [1] 1
#> [1] 2
#> [1] 3
`{`(print(1), print(2), print(3))
#> [1] 1
#> [1] 2
#> [1] 3
```

It is possible to override the definitions of these special functions, but this is almost certainly a bad idea. However, there are occasions when it might be useful: it allows you to do something that would have otherwise been impossible. For example, this feature makes it possible for the `dplyr` package to translate R expressions into SQL expressions. Chapter 15 uses this idea to create domain specific languages that allow you to concisely express new concepts using existing R constructs.

It's more often useful to treat special functions as ordinary functions. For example, we could use `sapply()` to add 3 to every element of a list by first defining a function `add()`, like this:

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
#>  [1]  4  5  6  7  8  9 10 11 12 13
```

But we can also get the same effect using the built-in `+` function.

```
sapply(1:5, `+`, 3)
#> [1] 4 5 6 7 8
sapply(1:5, "+", 3)
#> [1] 4 5 6 7 8
```

Note the difference between `` `+` `` and `"+"`. The first one is the value of the object called `+`, and the second is a string containing the character `+`. The

second version works because `sapply` can be given the name of a function instead of the function itself: if you read the source of `sapply()`, you'll see the first line uses `match.fun()` to find functions given their names.

A more useful application is to combine `lapply()` or `sapply()` with subsetting:

```
x <- list(1:3, 4:9, 10:12)
sapply(x, "[", 2)
#> [1]  2  5 11

# equivalent to
sapply(x, function(x) x[2])
#> [1]  2  5 11
```

Remembering that everything that happens in R is a function call will help you in Chapter 14.

6.4 Function 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. This section discusses how calling arguments are mapped to formal arguments, how you can call a function given a list of arguments, how default arguments work, and the impact of lazy evaluation.

6.4.1 Calling functions

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

```
f <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
```

```
str(f(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(f(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Can abbreviate long argument names:
str(f(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# But this doesn't work because abbreviation is ambiguous
str(f(1, 3, b = 1))
#> Error in f(1, 3, b = 1): argument 3 matches multiple formal arguments
```

Generally, you only want to use positional matching for the first one or two arguments; they will be the most commonly used, and most readers will know what they are. Avoid using positional matching for less commonly used arguments, and only use readable abbreviations with partial matching. (If you are writing code for a package that you want to publish on CRAN you can not use partial matching, and must use complete names.) Named arguments should always come after unnamed arguments. If a function uses `...` (discussed in more detail below), you can only specify arguments listed after `...` with their full name.

These are good calls:

```
mean(1:10)
mean(1:10, trim = 0.05)
```

This is probably overkill:

```
mean(x = 1:10)
```

And these are just confusing:

```
mean(1:10, n = T)
mean(1:10, , FALSE)
mean(1:10, 0.05)
mean(, TRUE, x = c(1:10, NA))
```

6.4.2 Calling a function given a list of arguments

Suppose you had a list of function arguments:

```
args <- list(1:10, na.rm = TRUE)
```

How could you then send that list to `mean()`? You need `do.call()`:

```
do.call(mean, list(1:10, na.rm = TRUE))
#> [1] 5.5
# Equivalent to
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

6.4.3 Default and missing arguments

Function arguments in R can have default values.

```
f <- function(a = 1, b = 2) {
  c(a, b)
}
f()
#> [1] 1 2
```

Since arguments in R are evaluated lazily (more on that below), the default value can be defined in terms of other arguments:

```
g <- function(a = 1, b = a * 2) {
  c(a, b)
}
g()
#> [1] 1 2
g(10)
#> [1] 10 20
```

Default arguments can even be defined in terms of variables created within the function. This is used frequently in base R functions, but I think it is bad practice, because you can't understand what the default values will be without reading the complete source code.

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
#> [1] 1 4
h(10)
#> [1]  10 121
```

You can determine if an argument was supplied or not with the `missing()` function.

```
i <- function(a, b) {
  c(missing(a), missing(b))
}
i()
#> [1] TRUE TRUE
i(a = 1)
#> [1] FALSE  TRUE
i(b = 2)
#> [1]  TRUE FALSE
i(1, 2)
#> [1] FALSE FALSE
```

Sometimes you want to add a non-trivial default value, which might take several lines of code to compute. Instead of inserting that code in the function definition, you could use `missing()` to conditionally compute it if needed. However, this makes it hard to know which arguments are required and which are optional without carefully reading the documentation. Instead, I usually set the default value to `NULL` and use `is.null()` to check if the argument was supplied.

6.4.4 Lazy evaluation

By default, R function arguments are lazy — they're only evaluated if they're actually used:

```
f <- function(x) {
  10
}
f(stop("This is an error!"))
#> [1] 10
```

If you want to ensure that an argument is evaluated you can use `force()`:

```
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))
#> Error in force(x): This is an error!
```

This is important when creating closures with `lapply()` or a loop:

```
add <- function(x) {
  function(y) x + y
}
adders <- lapply(1:10, add)
adders[[1]](10)
#> [1] 20
adders[[10]](10)
#> [1] 20
```

`x` is lazily evaluated the first time that you call one of the adder functions. At this point, the loop is complete and the final value of `x` is 10. Therefore all of the adder functions will add 10 on to their input, probably not what you wanted! Manually forcing evaluation fixes the problem:

```
add <- function(x) {
  force(x)
  function(y) x + y
}
adders2 <- lapply(1:10, add)
adders2[[1]](10)
#> [1] 11
adders2[[10]](10)
#> [1] 20
```

This code is exactly equivalent to

```
add <- function(x) {
  x
  function(y) x + y
}
```

because the force function is defined as `force <- function(x) x`. However, using this function clearly indicates that you're forcing evaluation, not that you've accidentally typed `x`.

Default arguments are evaluated inside the function. This means that if the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one.

```
f <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside f:
f()
#> [1] "a" "x"

# ls() evaluated in global environment:
f(ls())
#>  [1] "add"     "adders"  "adders2" "args"    "f"
#>  [6] "funs"    "g"       "h"       "i"       "objs"
#> [11] "x"       "y"
```

More technically, an unevaluated argument is called a **promise**, or (less commonly) a thunk. A promise is made up of two parts:

- The expression which gives rise to the delayed computation. (It can be accessed with `substitute()`. See Chapter 13 for more details.)
- The environment where the expression was created and where it should be evaluated.

The first time a promise is accessed the expression is evaluated in the environment where it was created. This value is cached, so that subsequent

access to the evaluated promise does not recompute the value (but the original expression is still associated with the value, so `substitute()` can continue to access it). You can find more information about a promise using `pryr::promise_info()`. This uses some C++ code to extract information about the promise without evaluating it, which is impossible to do in pure R code.

Laziness is useful in if statements — the second statement below will be evaluated only if the first is true. If it wasn't, the statement would return an error because `NULL > 0` is a logical vector of length 0 and not a valid input to `if`.

```
x <- NULL
if (!is.null(x) && x > 0) {

}
```

We could implement "&&" ourselves:

```
`&&` <- function(x, y) {
  if (!x) return(FALSE)
  if (!y) return(FALSE)

  TRUE
}
a <- NULL
!is.null(a) && a > 0
#> [1] FALSE
```

This function would not work without lazy evaluation because both `x` and `y` would always be evaluated, testing `a > 0` even when `a` was NULL.

Sometimes you can also use laziness to eliminate an if statement altogether. For example, instead of:

```
if (is.null(a)) stop("a is null")
#> Error in eval(expr, envir, enclos): a is null
```

You could write:

```
!is.null(a) || stop("a is null")
#> Error in eval(expr, envir, enclos): a is null
```

6.4.5 ...

There is a special argument called ... . This argument will match any arguments not otherwise matched, and can be easily passed on to other functions. This is useful if you want to collect arguments to call another function, but you don't want to prespecify their possible names. ... is often used in conjunction with S3 generic functions to allow individual methods to be more flexible.

One relatively sophisticated user of ... is the base `plot()` function. `plot()` is a generic method with arguments `x`, `y` and ... . To understand what ... does for a given function we need to read the help: "Arguments to be passed to methods, such as graphical parameters". Most simple invocations of `plot()` end up calling `plot.default()` which has many more arguments, but also has ... . Again, reading the documentation reveals that ... accepts "other graphical parameters", which are listed in the help for `par()`. This allows us to write code like:

```
plot(1:5, col = "red")
plot(1:5, cex = 5, pch = 20)
```

This illustrates both the advantages and disadvantages of ...: it makes `plot()` very flexible, but to understand how to use it, we have to carefully read the documentation. Additionally, if we read the source code for `plot.default`, we can discover undocumented features. It's possible to pass along other arguments to `Axis()` and `box()`:

```
plot(1:5, bty = "u")
plot(1:5, labels = FALSE)
```

To capture ... in a form that is easier to work with, you can use `list(...)`. (See Section 13.5.2 for other ways to capture ... without evaluating the arguments.)

```
f <- function(...) {
  names(list(...))
}
f(a = 1, b = 2)
#> [1] "a" "b"
```

Using ... comes at a price — any misspelled arguments will not raise an error, and any arguments after ... must be fully named. This makes it easy for typos to go unnoticed:

```
sum(1, 2, NA, na.mr = TRUE)
#> [1] NA
```

It's often better to be explicit rather than implicit, so you might instead ask users to supply a list of additional arguments. That's certainly easier if you're trying to use ... with multiple additional functions.

6.4.6 Exercises

1. Clarify the following list of odd function calls:

   ```
   x <- sample(replace = TRUE, 20, x = c(1:10, NA))
   y <- runif(min = 0, max = 1, 20)
   cor(m = "k", y = y, u = "p", x = x)
   ```

2. What does this function return? Why? Which principle does it illustrate?

   ```
   f1 <- function(x = {y <- 1; 2}, y = 0) {
     x + y
   }
   f1()
   ```

3. What does this function return? Why? Which principle does it illustrate?

   ```
   f2 <- function(x = z) {
     z <- 100
     x
   }
   f2()
   ```

6.5 Special calls

R supports two additional syntaxes for calling special types of functions: infix and replacement functions.

6.5.1 Infix functions

Most functions in R are "prefix" operators: the name of the function comes before the arguments. You can also create infix functions where the function name comes in between its arguments, like `+` or `-`. All user-created infix functions must start and end with `%`. R comes with the following infix functions predefined: `%%`, `%*%`, `%/%`, `%in%`, `%o%`, `%x%`. (The complete list of built-in infix operators that don't need `%` is: `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, `<<-`)

For example, we could create a new operator that pastes together strings:

```
`%+%` <- function(a, b) paste0(a, b)
"new" %+% " string"
#> [1] "new string"
```

Note that when creating the function, you have to put the name in backticks because it's a special name. This is just a syntactic sugar for an ordinary function call; as far as R is concerned there is no difference between these two expressions:

```
"new" %+% " string"
#> [1] "new string"
`%+%`("new", " string")
#> [1] "new string"
```

Or indeed between

```
1 + 5
#> [1] 6
`+`(1, 5)
#> [1] 6
```

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters (except "%", of course). You will need to escape any special characters in the string used to define the function, but not when you call it:

```
`% %` <- function(a, b) paste(a, b)
`%'%` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)
```

```
"a" %%% "b"
#> [1] "a b"
"a" %'% "b"
#> [1] "a b"
"a" %/\% "b"
#> [1] "a b"
```

R's default precedence rules mean that infix operators are composed from left to right:

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
#> [1] "((a %-% b) %-% c)"
```

There's one infix function that I use very often. It's inspired by Ruby's `||` logical or operator, although it works a little differently in R because Ruby has a more flexible definition of what evaluates to `TRUE` in an if statement. It's useful as a way of providing a default value in case the output of another function is `NULL`:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

6.5.2 Replacement functions

Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`. They typically have two arguments (`x` and `value`), although they can have more, and they must return the modified object. For example, the following function allows you to modify the second element of a vector:

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
x
#>  [1]  1  5  3  4  5  6  7  8  9 10
```

When R evaluates the assignment `second(x) <- 5`, it notices that the left hand side of the `<-` is not a simple name, so it looks for a function named `second<-` to do the replacement.

I say they "act" like they modify their arguments in place, because they actually create a modified copy. We can see that by using `pryr::address()` to find the memory address of the underlying object.

```
library(pryr)
x <- 1:10
address(x)
#> [1] "0x7fd589eda7a8"
second(x) <- 6L
address(x)
#> [1] "0x7fd58cb27678"
```

Built-in functions that are implemented using `.Primitive()` will modify in place:

```
x <- 1:10
address(x)
#> [1] "0x103945110"

x[2] <- 7L
address(x)
#> [1] "0x103945110"
```

It's important to be aware of this behaviour since it has important performance implications.

If you want to supply additional arguments, they go in between `x` and `value`:

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
#>  [1] 10  6  3  4  5  6  7  8  9 10
```

When you call `modify(x, 1) <- 10`, behind the scenes R turns it into:

```
x <- `modify<-`(x, 1, 10)
```

This means you can't do things like:

```
modify(get("x"), 1) <- 10
```

because that gets turned into the invalid code:

```
get("x") <- `modify<-`(get("x"), 1, 10)
```

It's often useful to combine replacement and subsetting:

```
x <- c(a = 1, b = 2, c = 3)
names(x)
#> [1] "a" "b" "c"
names(x)[2] <- "two"
names(x)
#> [1] "a"   "two" "c"
```

This works because the expression `names(x)[2] <- "two"` is evaluated as if you had written:

```
`*tmp*` <- names(x)
`*tmp*`[2] <- "two"
names(x) <- `*tmp*`
```

(Yes, it really does create a local variable named `*tmp*`, which is removed afterwards.)

6.5.3 Exercises

1. Create a list of all the replacement functions found in the base package. Which ones are primitive functions?
2. What are valid names for user-created infix functions?
3. Create an infix `xor()` operator.
4. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`.
5. Create a replacement function that modifies a random location in a vector.

6.6 Return values

The last expression evaluated in a function becomes the return value, the result of invoking the function.

```
f <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
#> [1] 0
f(15)
#> [1] 10
```

Generally, I think it's good style to reserve the use of an explicit `return()` for when you are returning early, such as for an error, or a simple case of the function. This style of programming can also reduce the level of indentation, and generally make functions easier to understand because you can reason about them locally.

```
f <- function(x, y) {
  if (!x) return(y)

  # complicated processing here
}
```

Functions can return only a single object. But this is not a limitation because you can return a list containing any number of objects.

The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no **side effects**: they don't affect the state of the world in any way apart from the value they return.

R protects you from one type of side effect: most R objects have copy-on-modify semantics. So modifying a function argument does not change the original value:

```
f <- function(x) {
  x$a <- 2
  x
}
x <- list(a = 1)
f(x)
#> $a
#> [1] 2
x$a
#> [1] 1
```

(There are two important exceptions to the copy-on-modify rule: environments and reference classes. These can be modified in place, so extra care is needed when working with them.)

This is notably different to languages like Java where you can modify the inputs of a function. This copy-on-modify behaviour has important performance consequences which are discussed in depth in **??**. (Note that the performance consequences are a result of R's implementation of copy-on-modify semantics; they are not true in general. Clojure is a new language that makes extensive use of copy-on-modify semantics with limited performance consequences.)

Most base R functions are pure, with a few notable exceptions:

- `library()` which loads a package, and hence modifies the search path.
- `setwd()`, `Sys.setenv()`, `Sys.setlocale()` which change the working directory, environment variables, and the locale, respectively.
- `plot()` and friends which produce graphical output.
- `write()`, `write.csv()`, `saveRDS()`, etc. which save output to disk.
- `options()` and `par()` which modify global settings.
- S4 related functions which modify global tables of classes and methods.
- Random number generators which produce different numbers each time you run them.

It's generally a good idea to minimise the use of side effects, and where possible, to minimise the footprint of side effects by separating pure from impure functions. Pure functions are easier to test (because all you need to worry about are the input values and the output), and are less likely

to work differently on different versions of R or on different platforms. For example, this is one of the motivating principles of ggplot2: most operations work on an object that represents a plot, and only the final `print` or `plot` call has the side effect of actually drawing the plot.

Functions can return `invisible` values, which are not printed out by default when you call the function.

```
f1 <- function() 1
f2 <- function() invisible(1)

f1()
#> [1] 1
f2()
f1() == 1
#> [1] TRUE
f2() == 1
#> [1] TRUE
```

You can force an invisible value to be displayed by wrapping it in parentheses:

```
(f2())
#> [1] 1
```

The most common function that returns invisibly is `<-`:

```
a <- 2
(a <- 2)
#> [1] 2
```

This is what makes it possible to assign one value to multiple variables:

```
a <- b <- c <- d <- 2
```

because this is parsed as:

```
(a <- (b <- (c <- (d <- 2))))
#> [1] 2
```

6.6.1 On exit

As well as returning a value, functions can set up other triggers to occur when the function is finished using `on.exit()`. This is often used as a way to guarantee that changes to the global state are restored when the function exits. The code in `on.exit()` is run regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body.

```
in_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old))

  force(code)
}
getwd()
#> [1] "/Users/hadley/Documents/adv-r/adv-r"
in_dir("~", getwd())
#> [1] "/Users/hadley"
```

The basic pattern is simple:

- We first set the directory to a new location, capturing the current location from the output of `setwd()`.
- We then use `on.exit()` to ensure that the working directory is returned to the previous value regardless of how the function exits.
- Finally, we explicitly force evaluation of the code. (We don't actually need `force()` here, but it makes it clear to readers what we're doing.)

Caution: If you're using multiple `on.exit()` calls within a function, make sure to set `add = TRUE`. Unfortunately, the default in `on.exit()` is `add = FALSE`, so that every time you run it, it overwrites existing exit expressions. Because of the way `on.exit()` is implemented, it's not possible to create a variant with `add = TRUE`, so you must be careful when using it.

6.6.2 Exercises

1. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?

2. What function undoes the action of `library()`? How do you save and restore the values of `options()` and `par()`?
3. Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code worked).
4. We can use `on.exit()` to implement a simple version of `capture.output()`.

   ```
   capture.output2 <- function(code) {
     temp <- tempfile()
     on.exit(file.remove(temp), add = TRUE)

     sink(temp)
     on.exit(sink(), add = TRUE)

     force(code)
     readLines(temp)
   }
   capture.output2(cat("a", "b", "c", sep = "\n"))
   #> [1] "a" "b" "c"
   ```

   Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas to be easier to understand?

6.7 Quiz answers

1. The three components of a function are its body, arguments, and environment.
2. `f1(1)()` returns 11.
3. You'd normally write it in infix style: `1 + (2 * 3)`.
4. Rewriting the call to `mean(c(1:10, NA), na.rm = TRUE)` is easier to understand.
5. No, it does not throw an error because the second argument is never used so it's never evaluated.
6. See Section 6.5.1 and Section 6.5.2.
7. You use `on.exit()`; see Section 6.6.1 for details.

7

OO field guide

This chapter is a field guide for recognising and working with R's objects in the wild. R has three object oriented systems (plus the base types), so it can be a bit intimidating. The goal of this guide is not to make you an expert in all four systems, but to help you identify which system you're working with and to help you use it effectively.

Central to any object-oriented system are the concepts of class and method. A **class** defines the behaviour of **objects** by describing their attributes and their relationship to other classes. The class is also used when selecting **methods**, functions that behave differently depending on the class of their input. Classes are usually organised in a hierarchy: if a method does not exist for a child, then the parent's method is used instead; the child **inherits** behaviour from the parent.

R's three OO systems differ in how classes and methods are defined:

- **S3** implements a style of OO programming called generic-function OO. This is different from most programming languages, like Java, C++, and C#, which implement message-passing OO. With message-passing, messages (methods) are sent to objects and the object determines which function to call. Typically, this object has a special appearance in the method call, usually appearing before the name of the method/message: e.g., `canvas.drawRect("blue")`. S3 is different. While computations are still carried out via methods, a special type of function called a **generic function** decides which method to call, e.g., `drawRect(canvas, "blue")`. S3 is a very casual system. It has no formal definition of classes.

- **S4** works similarly to S3, but is more formal. There are two major differences to S3. S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods. S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.

- **Reference classes**, called RC for short, are quite different from S3 and S4. RC implements message-passing OO, so methods belong to classes, not functions. `$` is used to separate objects and methods, so method calls look like `canvas$drawRect("blue")`. RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve with S3 or S4.

There's also one other system that's not quite OO, but it's important to mention here:

- **base types**, the internal C-level types that underlie the other OO systems. Base types are mostly manipulated using C code, but they're important to know about because they provide the building blocks for the other OO systems.

The following sections describe each system in turn, starting with base types. You'll learn how to recognise the OO system that an object belongs to, how method dispatch works, and how to create new objects, classes, generics, and methods for that system. The chapter concludes with a few remarks on when to use each system.

Prerequisites

You'll need the pryr package, `install.packages("pryr")`, to access useful functions for examining OO properties.

Quiz

Think you know this material already? If you can answer the following questions correctly, you can safely skip this chapter. Find the answers at the end of the chapter in Section 7.6.

1. How do you tell what OO system (base, S3, S4, or RC) an object is associated with?
2. How do you determine the base type (like integer or list) of an object?
3. What is a generic function?
4. What are the main differences between S3 and S4? What are the main differences between S4 & RC?

Outline

- Section 7.1 teaches you about R's base object system. Only R-core can add new classes to this system, but it's important to know about because it underpins the three other systems.
- Section 7.2 shows you the basics of the S3 object system. It's the simplest and most commonly used OO system.
- Section 7.3 discusses the more formal and rigorous S4 system.
- Section 7.4 teaches you about R's newest OO system: reference classes, or RC for short.
- Section 7.5 advises on which OO system to use if you're starting a new project.

7.1 Base types

Underlying every R object is a C structure (or struct) that describes how that object is stored in memory. The struct includes the contents of the object, the information needed for memory management, and, most importantly for this section, a **type**. This is the **base type** of an R object. Base types are not really an object system because only the R core team can create new types. As a result, new base types are added very rarely: the most recent change, in 2011, added two exotic types that you never see in R, but are useful for diagnosing memory problems (`NEWSXP` and `FREESXP`). Prior to that, the last type added was a special base type for S4 objects (`S4SXP`) in 2005.

Chapter 2 explains the most common base types (atomic vectors and lists), but base types also encompass functions, environments, and other more exotic objects likes names, calls, and promises that you'll learn about later in the book. You can determine an object's base type with `typeof()`. Unfortunately the names of base types are not used consistently throughout R, and type and the corresponding "is" function may use different names:

```
# The type of a function is "closure"
f <- function() {}
typeof(f)
```

```
#> [1] "closure"
is.function(f)
#> [1] TRUE

# The type of a primitive function is "builtin"
typeof(sum)
#> [1] "builtin"
is.primitive(sum)
#> [1] TRUE
```

You may have heard of `mode()` and `storage.mode()`. I recommend ignoring these functions because they're just aliases of the names returned by `typeof()`, and exist solely for S compatibility. Read their source code if you want to understand exactly what they do.

Functions that behave differently for different base types are almost always written in C, where dispatch occurs using switch statements (e.g., `switch(TYPEOF(x))`). Even if you never write C code, it's important to understand base types because everything else is built on top of them: S3 objects can be built on top of any base type, S4 objects use a special base type, and RC objects are a combination of S4 and environments (another base type). To see if an object is a pure base type, i.e., it doesn't also have S3, S4, or RC behaviour, check that `is.object(x)` returns `FALSE`.

7.2 S3

S3 is R's first and simplest OO system. It is the only OO system used in the base and stats packages, and it's the most commonly used system in CRAN packages. S3 is informal and ad hoc, but it has a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO system.

7.2.1 Recognising objects, generic functions, and methods

Most objects that you encounter are S3 objects. But unfortunately there's no simple way to test if an object is an S3 object in base R. The closest you can come is `is.object(x) & !isS4(x)`, i.e., it's an object, but not S4. An easier way is to use `pryr::otype()`:

```
library(pryr)

df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)    # A data frame is an S3 class
#> [1] "S3"
otype(df$x)  # A numeric vector isn't
#> [1] "base"
otype(df$y)  # A factor is
#> [1] "S3"
```

In S3, methods belong to functions, called **generic functions**, or generics for short. S3 methods do not belong to objects or classes. This is different from most other programming languages, but is a legitimate OO style.

To determine if a function is an S3 generic, you can inspect its source code for a call to `UseMethod()`: that's the function that figures out the correct method to call, the process of **method dispatch**. Similar to `otype()`, pryr also provides `ftype()` which describes the object system, if any, associated with a function:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7fb0b6295878>
#> <environment: namespace:base>
ftype(mean)
#> [1] "s3"      "generic"
```

Some S3 generics, like `[`, `sum()`, and `cbind()`, don't call `UseMethod()` because they are implemented in C. Instead, they call the C functions `DispatchGroup()` or `DispatchOrEval()`. Functions that do method dispatch in C code are called **internal generics** and are documented in `?"internal generic"`. `ftype()` knows about these special cases too.

Given a class, the job of an S3 generic is to call the right S3 method. You can recognise S3 methods by their names, which look like `generic.class()`. For example, the Date method for the `mean()` generic is called `mean.Date()`, and the factor method for `print()` is called `print.factor()`.

This is the reason that most modern style guides discourage the use of `.` in function names: it makes them look like S3 methods. For example, is `t.test()` the `test` method for `t` objects? Similarly, the use of `.` in class

names can also be confusing: is `print.data.frame()` the `print()` method for `data.frames`, or the `print.data()` method for `frames`? `pryr::ftype()` knows about these exceptions, so you can use it to figure out if a function is an S3 method or generic:

```
ftype(t.data.frame) # data frame method for t()
#> [1] "s3"     "method"
ftype(t.test)       # generic function for t tests
#> [1] "s3"      "generic"
```

You can see all the methods that belong to a generic with `methods()`:

```
methods("mean")
#> [1] mean.Date     mean.default  mean.difftime mean.POSIXct
#> [5] mean.POSIXlt
methods("t.test")
#> [1] t.test.default* t.test.formula*
#>
#>    Non-visible functions are asterisked
```

(Apart from methods defined in the base package, most S3 methods will not be visible: use `getS3method()` to read their source code.)

You can also list all generics that have a method for a given class:

```
methods(class = "ts")
#>  [1] [.ts*            [<-.ts*          aggregate.ts
#>  [4] as.data.frame.ts cbind.ts*        cycle.ts*
#>  [7] diff.ts*         diffinv.ts*      kernapply.ts*
#> [10] lines.ts*        monthplot.ts*    na.omit.ts*
#> [13] Ops.ts*          plot.ts          print.ts*
#> [16] t.ts*            time.ts*         window.ts*
#> [19] window<-.ts*
#>
#>    Non-visible functions are asterisked
```

There's no way to list all S3 classes, as you'll learn in the following section.

7.2.2 Defining classes and creating objects

S3 is a simple and ad hoc system; it has no formal definition of a class. To make an object an instance of a class, you just take an existing base object and set the class attribute. You can do that during creation with `structure()`, or after the fact with `class<-()`:

```
# Create and assign class in one step
foo <- structure(list(), class = "foo")

# Create, then set class
foo <- list()
class(foo) <- "foo"
```

S3 objects are usually built on top of lists, or atomic vectors with attributes. (You can refresh your memory of attributes with Section 2.2.) You can also turn functions into S3 objects. Other base types are either rarely seen in R, or have unusual semantics that don't work well with attributes.

You can determine the class of any object using `class(x)`, and see if an object inherits from a specific class using `inherits(x, "classname")`.

```
class(foo)
#> [1] "foo"
inherits(foo, "foo")
#> [1] TRUE
```

The class of an S3 object can be a vector, which describes behaviour from most to least specific. For example, the class of the `glm()` object is `c("glm", "lm")` indicating that generalised linear models inherit behaviour from linear models. Class names are usually lower case, and you should avoid `.`. Otherwise, opinion is mixed whether to use underscores (`my_class`) or CamelCase (`MyClass`) for multi-word class names.

Most S3 classes provide a constructor function:

```
foo <- function(x) {
  if (!is.numeric(x)) stop("X must be numeric")
  structure(list(x), class = "foo")
}
```

You should use it if it's available (like for `factor()` and `data.frame()`). This ensures that you're creating the class with the correct components. Constructor functions usually have the same name as the class.

Apart from developer supplied constructor functions, S3 has no checks for correctness. This means you can change the class of existing objects:

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept)    log(disp)
#>       5.381       -0.459

# Turn it into a data frame (?!)
class(mod) <- "data.frame"
# But unsurprisingly this doesn't work very well
print(mod)
#>  [1] coefficients  residuals     effects       rank
#>  [5] fitted.values assign        qr            df.residual
#>  [9] xlevels       call          terms         model
#> <0 rows> (or 0-length row.names)
# However, the data is still there
mod$coefficients
#> (Intercept)   log(disp)
#>       5.381      -0.459
```

If you've used other OO languages, this might make you feel queasy. But surprisingly, this flexibility causes few problems: while you *can* change the type of an object, you never should. R doesn't protect you from yourself: you can easily shoot yourself in the foot. As long as you don't aim the gun at your foot and pull the trigger, you won't have a problem.

7.2.3 Creating new methods and generics

To add a new generic, create a function that calls `UseMethod()`. `UseMethod()` takes two arguments: the name of the generic function,

and the argument to use for method dispatch. If you omit the second argument it will dispatch on the first argument to the function. There's no need to pass any of the arguments of the generic to `UseMethod()` and you shouldn't do so. `UseMethod()` uses black magic to find them out for itself.

```
f <- function(x) UseMethod("f")
```

A generic isn't useful without some methods. To add a method, you just create a regular function with the correct (`generic.class`) name:

```
f.a <- function(x) "Class a"

a <- structure(list(), class = "a")
class(a)
#> [1] "a"
f(a)
#> [1] "Class a"
```

Adding a method to an existing generic works in the same way:

```
mean.a <- function(x) "a"
mean(a)
#> [1] "a"
```

As you can see, there's no check to make sure that the method returns the class compatible with the generic. It's up to you to make sure that your method doesn't violate the expectations of existing code.

7.2.4 Method dispatch

S3 method dispatch is relatively simple. `UseMethod()` creates a vector of function names, like `paste0("generic", ".", c(class(x), "default"))` and looks for each in turn. The "default" class makes it possible to set up a fall back method for otherwise unknown classes.

```
f <- function(x) UseMethod("f")
f.a <- function(x) "Class a"
f.default <- function(x) "Unknown class"
```

```
f(structure(list(), class = "a"))
#> [1] "Class a"
# No method for b class, so uses method for a class
f(structure(list(), class = c("b", "a")))
#> [1] "Class a"
# No method for c class, so falls back to default
f(structure(list(), class = "c"))
#> [1] "Unknown class"
```

Group generic methods add a little more complexity. Group generics make it possible to implement methods for multiple generics with one function. The four group generics and the functions they include are:

- Math: `abs`, `sign`, `sqrt`, `floor`, `cos`, `sin`, `log`, `exp`, …
- Ops: `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`, `&`, `|`, `!`, `==`, `!=`, `<`, `<=`, `>=`, `>`
- Summary: `all`, `any`, `sum`, `prod`, `min`, `max`, `range`
- Complex: `Arg`, `Conj`, `Im`, `Mod`, `Re`

Group generics are a relatively advanced technique and are beyond the scope of this chapter but you can find out more about them in `?groupGeneric`. The most important thing to take away from this is to recognise that `Math`, `Ops`, `Summary`, and `Complex` aren't real functions, but instead represent groups of functions. Note that inside a group generic function a special variable `.Generic` provides the actual generic function called.

If you have complex class hierarchies it's sometimes useful to call the "parent" method. It's a little bit tricky to define exactly what that means, but it's basically the method that would have been called if the current method did not exist. Again, this is an advanced technique: you can read about it in `?NextMethod`.

Because methods are normal R functions, you can call them directly:

```
c <- structure(list(), class = "c")
# Call the correct method:
f.default(c)
#> [1] "Unknown class"
# Force R to call the wrong method:
f.a(c)
#> [1] "Class a"
```

However, this is just as dangerous as changing the class of an object, so you shouldn't do it. Please don't point the loaded gun at your foot! The

only reason to call the method directly is that sometimes you can get considerable performance improvements by skipping method dispatch. See **??** for details.

You can also call an S3 generic with a non-S3 object. Non-internal S3 generics will dispatch on the **implicit class** of base types. (Internal generics don't do that for performance reasons.) The rules to determine the implicit class of a base type are somewhat complex, but are shown in the function below:

```
iclass <- function(x) {
  if (is.object(x)) {
    stop("x is not a primitive type", call. = FALSE)
  }

  c(
    if (is.matrix(x)) "matrix",
    if (is.array(x) && !is.matrix(x)) "array",
    if (is.double(x)) "double",
    if (is.integer(x)) "integer",
    mode(x)
  )
}
iclass(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
iclass(array(1.5))
#> [1] "array"   "double"  "numeric"
```

7.2.5 Exercises

1. Read the source code for `t()` and `t.test()` and confirm that `t.test()` is an S3 generic and not an S3 method. What happens if you create an object with class `test` and call `t()` with it?
2. What classes have a method for the `Math` group generic in base R? Read the source code. How do the methods work?
3. R has two classes for representing date time data, `POSIXct` and `POSIXlt`, which both inherit from `POSIXt`. Which generics have different behaviours for the two classes? Which generics share the same behaviour?
4. Which base generic has the greatest number of defined methods?

5. `UseMethod()` calls methods in a special way. Predict what the following code will return, then run it and read the help for `UseMethod()` to figure out what's going on. Write down the rules in the simplest form possible.

```
y <- 1
g <- function(x) {
  y <- 2
  UseMethod("g")
}
g.numeric <- function(x) y
g(10)

h <- function(x) {
  x <- 10
  UseMethod("h")
}
h.character <- function(x) paste("char", x)
h.numeric <- function(x) paste("num", x)

h("a")
```

6. Internal generics don't dispatch on the implicit class of base types. Carefully read `?"internal generic"` to determine why the length of `f` and `g` is different in the example below. What function helps distinguish between the behaviour of `f` and `g`?

```
f <- function() 1
g <- function() 2
class(g) <- "function"

class(f)
class(g)

length.function <- function(x) "function"
length(f)
length(g)
```

7.3 S4

S4 works in a similar way to S3, but it adds formality and rigour. Methods still belong to functions, not classes, but:

- Classes have formal definitions which describe their fields and inheritance structures (parent classes).
- Method dispatch can be based on multiple arguments to a generic function, not just one.
- There is a special operator, `@`, for extracting slots (aka fields) from an S4 object.

All S4 related code is stored in the `methods` package. This package is always available when you're running R interactively, but may not be available when running R in batch mode. For this reason, it's a good idea to include an explicit `library(methods)` whenever you're using S4.

S4 is a rich and complex system. There's no way to explain it fully in a few pages. Here I'll focus on the key ideas underlying S4 so you can use existing S4 objects effectively. To learn more, some good references are:

- S4 system development in Bioconductor (`http://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf`)
- John Chambers' *Software for Data Analysis* (`http://amzn.com/0387759352?tag=devtools-20`)
- Martin Morgan's answers to S4 questions on stackoverflow (`http://stackoverflow.com/search?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe`)

7.3.1 Recognising objects, generic functions, and methods

Recognising S4 objects, generics, and methods is easy. You can identify an S4 object because `str()` describes it as a "formal" class, `isS4()` returns `TRUE`, and `pryr::otype()` returns "S4". S4 generics and methods are also easy to identify because they are S4 objects with well defined classes.

There aren't any S4 classes in the commonly used base packages (stats, graphics, utils, datasets, and base), so we'll start by creating an S4 object from the built-in stats4 package, which provides some S4 classes and methods associated with maximum likelihood estimation:

```
library(stats4)

# From example(mle)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) - sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))

# An S4 object
isS4(fit)
#> [1] TRUE
otype(fit)
#> [1] "S4"

# An S4 generic
isS4(nobs)
#> [1] TRUE
ftype(nobs)
#> [1] "s4"      "generic"

# Retrieve an S4 method, described later
mle_nobs <- method_from_call(nobs(fit))
isS4(mle_nobs)
#> [1] TRUE
ftype(mle_nobs)
#> [1] "s4"     "method"
```

Use `is()` with one argument to list all classes that an object inherits from. Use `is()` with two arguments to test if an object inherits from a specific class.

```
is(fit)
#> [1] "mle"
is(fit, "mle")
#> [1] TRUE
```

You can get a list of all S4 generics with `getGenerics()`, and a list of all S4 classes with `getClasses()`. This list includes shim classes for S3

classes and base types. You can list all S4 methods with `showMethods()`, optionally restricting selection either by `generic` or by `class` (or both). It's also a good idea to supply `where = search()` to restrict the search to methods available in the global environment.

7.3.2 Defining classes and creating objects

In S3, you can turn any object into an object of a particular class just by setting the class attribute. S4 is much stricter: you must define the representation of a class with `setClass()`, and create a new object with `new()`. You can find the documentation for a class with a special syntax: `class?className`, e.g., `class?mle`.

An S4 class has three key properties:

- A **name**: an alpha-numeric class identifier. By convention, S4 class names use UpperCamelCase.
- A named list of **slots** (fields), which defines slot names and permitted classes. For example, a person class might be represented by a character name and a numeric age: `list(name = "character", age = "numeric")`.
- A string giving the class it inherits from, or, in S4 terminology, that it **contains**. You can provide multiple classes for multiple inheritance, but this is an advanced technique which adds much complexity.

In `slots` and `contains` you can use S4 classes, S3 classes registered with `setOldClass()`, or the implicit class of a base type. In `slots` you can also use the special class `ANY` which does not restrict the input.

S4 classes have other optional properties like a `validity` method that tests if an object is valid, and a `prototype` object that defines default slot values. See `?setClass` for more details.

The following example creates a Person class with fields name and age, and an Employee class that inherits from Person. The Employee class inherits the slots and methods from the Person, and adds an additional slot, boss. To create objects we call `new()` with the name of the class, and name-value pairs of slot values.

```
setClass("Person",
  slots = list(name = "character", age = "numeric"))
setClass("Employee",
```

```
  slots = list(boss = "Person"),
  contains = "Person")

alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)
```

Most S4 classes also come with a constructor function with the same name as the class: if that exists, use it instead of calling `new()` directly.

To access slots of an S4 object use `@` or `slot()`:

```
alice@age
#> [1] 40
slot(john, "boss")
#> An object of class "Person"
#> Slot "name":
#> [1] "Alice"
#>
#> Slot "age":
#> [1] 40
```

(`@` is equivalent to `$`, and `slot()` to `[[`.)

If an S4 object contains (inherits from) an S3 class or a base type, it will have a special `.Data` slot which contains the underlying base type or S3 object:

```
setClass("RangedNumeric",
  contains = "numeric",
  slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10, min = 1, max = 10)
rn@min
#> [1] 1
rn@.Data
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Since R is an interactive programming language, it's possible to create new classes or redefine existing classes at any time. This can be a problem when you're interactively experimenting with S4. If you modify a class, make sure you also recreate any objects of that class, otherwise you'll end up with invalid objects.

7.3.3 Creating new methods and generics

S4 provides special functions for creating new generics and methods. `setGeneric()` creates a new generic or converts an existing function into a generic. `setMethod()` takes the name of the generic, the classes the method should be associated with, and a function that implements the method. For example, we could take `union()`, which usually just works on vectors, and make it work with data frames:

```
setGeneric("union")
#> [1] "union"
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
#> [1] "union"
```

If you create a new generic from scratch, you need to supply a function that calls `standardGeneric()`:

```
setGeneric("myGeneric", function(x) {
  standardGeneric("myGeneric")
})
#> [1] "myGeneric"
```

`standardGeneric()` is the S4 equivalent to `UseMethod()`.

7.3.4 Method dispatch

If an S4 generic dispatches on a single class with a single parent, then S4 method dispatch is the same as S3 dispatch. The main difference is how you set up default values: S4 uses the special class `ANY` to match any class and "missing" to match a missing argument. Like S3, S4 also has group generics, documented in `?S4groupGeneric`, and a way to call the "parent" method, `callNextMethod()`.

Method dispatch becomes considerably more complicated if you dispatch on multiple arguments, or if your classes use multiple inheritance. The rules are described in `?Methods`, but they are complicated and it's difficult to predict which method will be called. For this reason, I strongly

recommend avoiding multiple inheritance and multiple dispatch unless absolutely necessary.

Finally, there are two methods that find which method gets called given the specification of a generic call:

```
# From methods: takes generic name and class names
selectMethod("nobs", list("mle"))

# From pryr: takes an unevaluated function call
method_from_call(nobs(fit))
```

7.3.5 Exercises

1. Which S4 generic has the most methods defined for it? Which S4 class has the most methods associated with it?
2. What happens if you define a new S4 class that doesn't "contain" an existing class? (Hint: read about virtual classes in `?Classes`.)
3. What happens if you pass an S4 object to an S3 generic? What happens if you pass an S3 object to an S4 generic? (Hint: read `?setOldClass` for the second case.)

7.4 RC

Reference classes (or RC for short) are the newest OO system in R. They were introduced in version 2.12. They are fundamentally different to S3 and S4 because:

- RC methods belong to objects, not functions
- RC objects are mutable: the usual R copy-on-modify semantics do not apply

These properties make RC objects behave more like objects do in most other programming languages, e.g., Python, Ruby, Java, and C#. Reference classes are implemented using R code: they are a special S4 class that wraps around an environment.

7.4.1 Defining classes and creating objects

Since there aren't any reference classes provided by the base R packages, we'll start by creating one. RC classes are best used for describing stateful objects, objects that change over time, so we'll create a simple class to model a bank account.

Creating a new RC class is similar to creating a new S4 class, but you use `setRefClass()` instead of `setClass()`. The first, and only required argument, is an alphanumeric **name**. While you can use `new()` to create new RC objects, it's good style to use the object returned by `setRefClass()` to generate new objects. (You can also do that with S4 classes, but it's less common.)

```
Account <- setRefClass("Account")
Account$new()
#> Reference class object of class "Account"
```

`setRefClass()` also accepts a list of name-class pairs that define class **fields** (equivalent to S4 slots). Additional named arguments passed to `new()` will set initial values of the fields. You can get and set field values with `$`:

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"))

a <- Account$new(balance = 100)
a$balance
#> [1] 100
a$balance <- 200
a$balance
#> [1] 200
```

Instead of supplying a class name for the field, you can provide a single argument function which will act as an accessor method. This allows you to add custom behaviour when getting or setting a field. See `?setRefClass` for more details.

Note that RC objects are **mutable**, i.e., they have reference semantics, and are not copied-on-modify:

```
b <- a
b$balance
```

```
#> [1] 200
a$balance <- 0
b$balance
#> [1] 0
```

For this reason, RC objects come with a `copy()` method that allow you to make a copy of the object:

```
c <- a$copy()
c$balance
#> [1] 0
a$balance <- 100
c$balance
#> [1] 0
```

An object is not very useful without some behaviour defined by **methods**. RC methods are associated with a class and can modify its fields in place. In the following example, note that you access the value of fields with their name, and modify them with `<<-`. You'll learn more about `<<-` in Section 8.4.

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      balance <<- balance - x
    },
    deposit = function(x) {
      balance <<- balance + x
    }
  )
)
```

You call an RC method in the same way as you access a field:

```
a <- Account$new(balance = 100)
a$deposit(100)
a$balance
#> [1] 200
```

The final important argument to `setRefClass()` is `contains`. This is the

name of the parent RC class to inherit behaviour from. The following example creates a new type of bank account that returns an error preventing the balance from going below 0.

```
NoOverdraft <- setRefClass("NoOverdraft",
  contains = "Account",
  methods = list(
    withdraw = function(x) {
      if (balance < x) stop("Not enough money")
      balance <<- balance - x
    }
  )
)
accountJohn <- NoOverdraft$new(balance = 100)
accountJohn$deposit(50)
accountJohn$balance
#> [1] 150
accountJohn$withdraw(200)
#> Error in accountJohn$withdraw(200): Not enough money
```

All reference classes eventually inherit from `envRefClass`. It provides useful methods like `copy()` (shown above), `callSuper()` (to call the parent field), `field()` (to get the value of a field given its name), `export()` (equivalent to `as()`), and `show()` (overridden to control printing). See the inheritance section in `setRefClass()` for more details.

7.4.2 Recognising objects and methods

You can recognise RC objects because they are S4 objects (`isS4(x)`) that inherit from "refClass" (`is(x, "refClass")`). `pryr::otype()` will return "RC". RC methods are also S4 objects, with class `refMethodDef`.

7.4.3 Method dispatch

Method dispatch is very simple in RC because methods are associated with classes, not functions. When you call `x$f()`, R will look for a method f in the class of x, then in its parent, then its parent's parent, and so on. From within a method, you can call the parent method directly with `callSuper(...)`.

7.4.4 Exercises

1. Use a field function to prevent the account balance from being directly manipulated. (Hint: create a "hidden" `.balance` field, and read the help for the fields argument in `setRefClass()`.)
2. I claimed that there aren't any RC classes in base R, but that was a bit of a simplification. Use `getClasses()` and find which classes `extend()` from `envRefClass`. What are the classes used for? (Hint: recall how to look up the documentation for a class.)

7.5 Picking a system

Three OO systems is a lot for one language, but for most R programming, S3 suffices. In R you usually create fairly simple objects and methods for pre-existing generic functions like `print()`, `summary()`, and `plot()`. S3 is well suited to this task, and the majority of OO code that I have written in R is S3. S3 is a little quirky, but it gets the job done with a minimum of code.

If you are creating more complicated systems of interrelated objects, S4 may be more appropriate. A good example is the `Matrix` package by Douglas Bates and Martin Maechler. It is designed to efficiently store and compute with many different types of sparse matrices. As of version 1.1.3, it defines 102 classes and 20 generic functions. The package is well written and well commented, and the accompanying vignette (`vignette("Intro2Matrix", package = "Matrix")`) gives a good overview of the structure of the package. S4 is also used extensively by Bioconductor packages, which need to model complicated interrelationships between biological objects. Bioconductor provides many good resources (`https://www.google.com/search?q=bioconductor+s4`) for learning S4. If you've mastered S3, S4 is relatively easy to pick up; the ideas are all the same, it is just more formal, more strict, and more verbose.

If you've programmed in a mainstream OO language, RC will seem very natural. But because they can introduce side effects through mutable state, they are harder to understand. For example, when you usually call `f(a, b)` in R you can assume that `a` and `b` will not be modified. But if `a` and `b` are RC objects, they might be modified in the place. Generally, when using RC objects you want to minimise side effects as

much as possible, and use them only where mutable states are absolutely required. The majority of functions should still be "functional", and free of side effects. This makes code easier to reason about and easier for other R programmers to understand.

7.6 Quiz answers

1. To determine the OO system of an object, you use a process of elimination. If `!is.object(x)`, it's a base object. If `!isS4(x)`, it's S3. If `!is(x, "refClass")`, it's S4; otherwise it's RC.
2. Use `typeof()` to determine the base class of an object.
3. A generic function calls specific methods depending on the class of it inputs. In S3 and S4 object systems, methods belong to generic functions, not classes like in other programming languages.
4. S4 is more formal than S3, and supports multiple inheritance and multiple dispatch. RC objects have reference semantics, and methods belong to classes, not functions.

8

Environments

The environment is the data structure that powers scoping. This chapter dives deep into environments, describing their structure in depth, and using them to improve your understanding of the four scoping rules described in Section 6.2.

Environments can also be useful data structures in their own right because they have reference semantics. When you modify a binding in an environment, the environment is not copied; it's modified in place. Reference semantics are not often needed, but can be extremely useful.

Quiz

If you can answer the following questions correctly, you already know the most important topics in this chapter. You can find the answers at the end of the chapter in Section 8.6.

1. List at least three ways that an environment is different to a list.
2. What is the parent of the global environment? What is the only environment that doesn't have a parent?
3. What is the enclosing environment of a function? Why is it important?
4. How do you determine the environment from which a function was called?
5. How are `<-` and `<<-` different?

Outline

- Section 8.1 introduces you to the basic properties of an environment and shows you how to create your own.
- Section 8.2 provides a function template for computing with environments, illustrating the idea with a useful function.

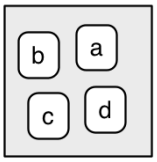
- Section 8.3 revises R's scoping rules in more depth, showing how they correspond to four types of environment associated with each function.
- Section 8.4 describes the rules that names must follow (and how to bend them), and shows some variations on binding a name to a value.
- Section 8.5 discusses three problems where environments are useful data structures in their own right, independent of the role they place in scoping.

Prerequisites

This chapter uses many functions from the `pryr` package to pry open R and look inside at the messy details. You can install `pryr` by running `install.packages("pryr")`

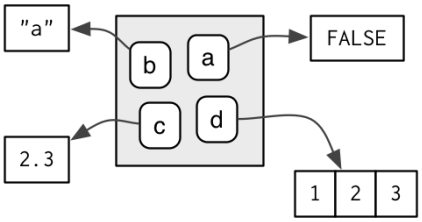
8.1 Environment basics

The job of an environment is to associate, or **bind**, a set of names to a set of values. You can think of an environment as a bag of names:



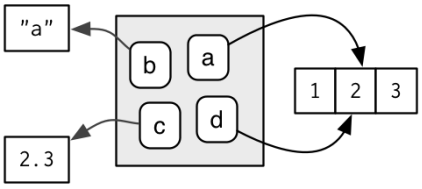
Each name points to an object stored elsewhere in memory:

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```



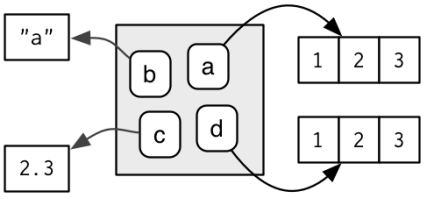
The objects don't live in the environment so multiple names can point to the same object:

```
e$a <- e$d
```



Confusingly they can also point to different objects that have the same value:

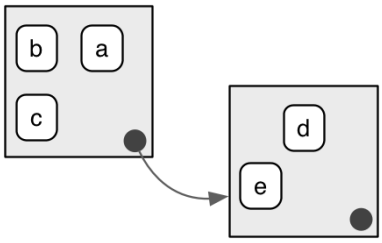
```
e$a <- 1:3
```



If an object has no names pointing to it, it gets automatically deleted by the garbage collector. This process is described in more detail in **??**.

Every environment has a parent, another environment. In diagrams, I'll

represent the pointer to parent with a small black circle. The parent is used to implement lexical scoping: if a name is not found in an environment, then R will look in its parent (and so on). Only one environment doesn't have a parent: the **empty** environment.



We use the metaphor of a family to refer to environments. The grandparent of an environment is the parent's parent, and the ancestors include all parent environments up to the empty environment. It's rare to talk about the children of an environment because there are no back links: given an environment we have no way to find its children.

Generally, an environment is similar to a list, with four important exceptions:

- Every object in an environment has a unique name.
- The objects in an environment are not ordered (i.e., it doesn't make sense to ask what the first object in an environment is).
- An environment has a parent.
- Environments have reference semantics.

More technically, an environment is made up of two components, the **frame**, which contains the name-object bindings (and behaves much like a named list), and the parent environment. Unfortunately "frame" is used inconsistently in R. For example, `parent.frame()` doesn't give you the parent frame of an environment. Instead, it gives you the *calling* environment. This is discussed in more detail in Section 8.3.4.

There are four special environments:

- The `globalenv()`, or global environment, is the interactive workspace. This is the environment in which you normally work. The parent of the global environment is the last package that you attached with `library()` or `require()`.

- The `baseenv()`, or base environment, is the environment of the base package. Its parent is the empty environment.
- The `emptyenv()`, or empty environment, is the ultimate ancestor of all environments, and the only environment without a parent.
- The `environment()` is the current environment.

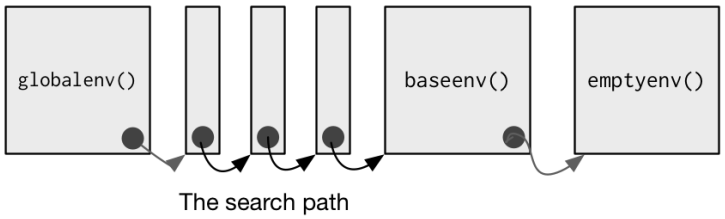
`search()` lists all parents of the global environment. This is called the search path because objects in these environments can be found from the top-level interactive workspace. It contains one environment for each attached package and any other objects that you've `attach()`ed. It also contains a special environment called `Autoloads` which is used to save memory by only loading package objects (like big datasets) when needed.

You can access any environment on the search list using `as.environment()`.

```
search()
#> [1] ".GlobalEnv"        "package:stats"     "package:graphics"
#> [4] "package:grDevices" "package:utils"     "package:datasets"
#> [7] "package:methods"   "Autoloads"         "package:base"

as.environment("package:stats")
#> <environment: package:stats>
```

`globalenv()`, `baseenv()`, the environments on the search path, and `emptyenv()` are connected as shown below. Each time you load a new package with `library()` it is inserted between the global environment and the package that was previously at the top of the search path.



To create an environment manually, use `new.env()`. You can list the bindings in the environment's frame with `ls()` and see its parent with `parent.env()`.

```
e <- new.env()
# the default parent provided by new.env() is environment from
# which it is called - in this case that's the global environment.
parent.env(e)
#> <environment: R_GlobalEnv>
ls(e)
#> character(0)
```

The easiest way to modify the bindings in an environment is to treat it like a list:

```
e$a <- 1
e$b <- 2
ls(e)
#> [1] "a" "b"
e$a
#> [1] 1
```

By default, `ls()` only shows names that don't begin with `.`. Use `all.names = TRUE` to show all bindings in an environment:

```
e$.a <- 2
ls(e)
#> [1] "a" "b"
ls(e, all.names = TRUE)
#> [1] ".a" "a"  "b"
```

Another useful way to view an environment is `ls.str()`. It is more useful than `str()` because it shows each object in the environment. Like `ls()`, it also has an `all.names` argument.

```
str(e)
#> <environment: 0x7fba6aac1ad8>
ls.str(e)
#> a :  num 1
#> b :  num 2
```

Given a name, you can extract the value to which it is bound with `$`, `[[`, or `get()`:

- `$` and `[[` look only in one environment and return `NULL` if there is no binding associated with the name.

- `get()` uses the regular scoping rules and throws an error if the binding is not found.

```
e$c <- 3
e$c
#> [1] 3
e[["c"]]
#> [1] 3
get("c", envir = e)
#> [1] 3
```

Deleting objects from environments works a little differently from lists. With a list you can remove an entry by setting it to `NULL`. In environments, that will create a new binding to `NULL`. Instead, use `rm()` to remove the binding.

```
e <- new.env()

e$a <- 1
e$a <- NULL
ls(e)
#> [1] "a"

rm("a", envir = e)
ls(e)
#> character(0)
```

You can determine if a binding exists in an environment with `exists()`. Like `get()`, its default behaviour is to follow the regular scoping rules and look in parent environments. If you don't want this behavior, use `inherits = FALSE`:

```
x <- 10
exists("x", envir = e)
#> [1] TRUE
exists("x", envir = e, inherits = FALSE)
#> [1] FALSE
```

To compare environments, you must use `identical()` not `==`:

```
identical(globalenv(), environment())
#> [1] TRUE
globalenv() == environment()
#> Error in globalenv() == environment(): comparison (1) is possible only for atomic and list typ
```

8.1.1 Exercises

1. List three ways in which an environment differs from a list.
2. If you don't supply an explicit environment, where do `ls()` and `rm()` look? Where does `<-` make bindings?
3. Using `parent.env()` and a loop (or a recursive function), verify that the ancestors of `globalenv()` include `baseenv()` and `emptyenv()`. Use the same basic idea to implement your own version of `search()`.

8.2 Recursing over environments

Environments form a tree, so it's often convenient to write a recursive function. This section shows you how by applying your new knowledge of environments to understand the helpful `pryr::where()`. Given a name, `where()` finds the environment *where* that name is defined, using R's regular scoping rules:

```
library(pryr)
x <- 5
where("x")
#> <environment: R_GlobalEnv>
where("mean")
#> <environment: base>
```

The definition of `where()` is straightforward. It has two arguments: the name to look for (as a string), and the environment in which to start the search. (We'll learn later why `parent.frame()` is a good default in Section 8.3.4.)

```
where <- function(name, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # Base case
    stop("Can't find ", name, call. = FALSE)

  } else if (exists(name, envir = env, inherits = FALSE)) {
    # Success case
    env
```

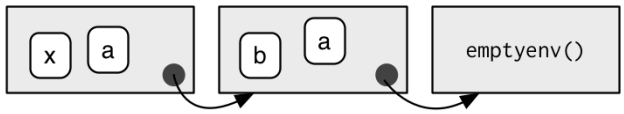
```
  } else {
    # Recursive case
    where(name, parent.env(env))

  }
}
```

There are three cases:

- The base case: we've reached the empty environment and haven't found the binding. We can't go any further, so we throw an error.
- The successful case: the name exists in this environment, so we return the environment.
- The recursive case: the name was not found in this environment, so try the parent.

It's easier to see what's going on with an example. Imagine you have two environments as in the following diagram:



- If you're looking for `a`, `where()` will find it in the first environment.
- If you're looking for `b`, it's not in the first environment, so `where()` will look in its parent and find it there.
- If you're looking for `c`, it's not in the first environment, or the second environment, so `where()` reaches the empty environment and throws an error.

It's natural to work with environments recursively, so `where()` provides a useful template. Removing the specifics of `where()` shows the structure more clearly:

```
f <- function(..., env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # base case
```

```
  } else if (success) {
    # success case
  } else {
    # recursive case
    f(..., env = parent.env(env))
  }
}
```

Iteration vs. recursion

It's possible to use a loop instead of recursion. This might run slightly faster (because we eliminate some function calls), but I think it's harder to understand. I include it because you might find it easier to see what's happening if you're less familiar with recursive functions.

```
is_empty <- function(x) identical(x, emptyenv())

f2 <- function(..., env = parent.frame()) {
  while(!is_empty(env)) {
    if (success) {
      # success case
      return()
    }
    # inspect parent
    env <- parent.env(env)
  }

  # base case
}
```

8.2.1 Exercises

1. Modify `where()` to find all environments that contain a binding for `name`.
2. Write your own version of `get()` using a function written in the style of `where()`.
3. Write a function called `fget()` that finds only function objects. It should have two arguments, `name` and `env`, and should obey the regular scoping rules for functions: if there's an object with

a matching name that's not a function, look in the parent. For an added challenge, also add an `inherits` argument which controls whether the function recurses up the parents or only looks in one environment.

4. Write your own version of `exists(inherits = FALSE)` (Hint: use `ls()`.) Write a recursive version that behaves like `exists(inherits = TRUE)`.

8.3 Function environments

Most environments are not created by you with `new.env()` but are created as a consequence of using functions. This section discusses the four types of environments associated with a function: enclosing, binding, execution, and calling.

The **enclosing** environment is the environment where the function was created. Every function has one and only one enclosing environment. For the three other types of environment, there may be 0, 1, or many environments associated with each function:

- Binding a function to a name with `<-` defines a **binding** environment.
- Calling a function creates an ephemeral **execution** environment that stores variables created during execution.
- Every execution environment is associated with a **calling** environment, which tells you where the function was called.

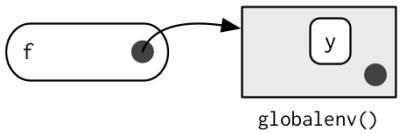
The following sections will explain why each of these environments is important, how to access them, and how you might use them.

8.3.1 The enclosing environment

When a function is created, it gains a reference to the environment where it was made. This is the **enclosing environment** and is used for lexical scoping. You can determine the enclosing environment of a function by calling `environment()` with a function as its first argument:

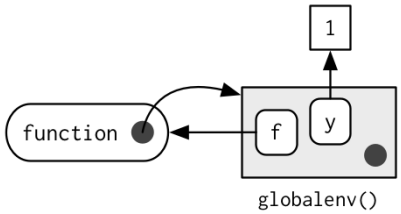
```
y <- 1
f <- function(x) x + y
environment(f)
#> <environment: R_GlobalEnv>
```

In diagrams, I'll depict functions as rounded rectangles. The enclosing environment of a function is given by a small black circle:



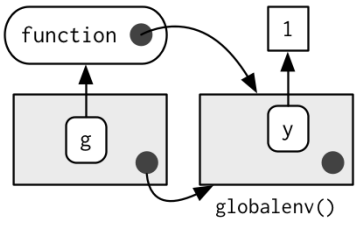
8.3.2 Binding environments

The previous diagram is too simple because functions don't have names. Instead, the name of a function is defined by a binding. The binding environments of a function are all the environments which have a binding to it. The following diagram better reflects this relationship because the enclosing environment contains a binding from `f` to the function:



In this case the enclosing and binding environments are the same. They will be different if you assign a function into a different environment:

```
e <- new.env()
e$g <- function() 1
```



The enclosing environment belongs to the function, and never changes, even if the function is moved to a different environment. The enclosing environment determines how the function finds values; the binding environments determine how we find the function.

The distinction between the binding environment and the enclosing environment is important for package namespaces. Package namespaces keep packages independent. For example, if package A uses the base `mean()` function, what happens if package B creates its own `mean()` function? Namespaces ensure that package A continues to use the base `mean()` function, and that package A is not affected by package B (unless explicitly asked for).

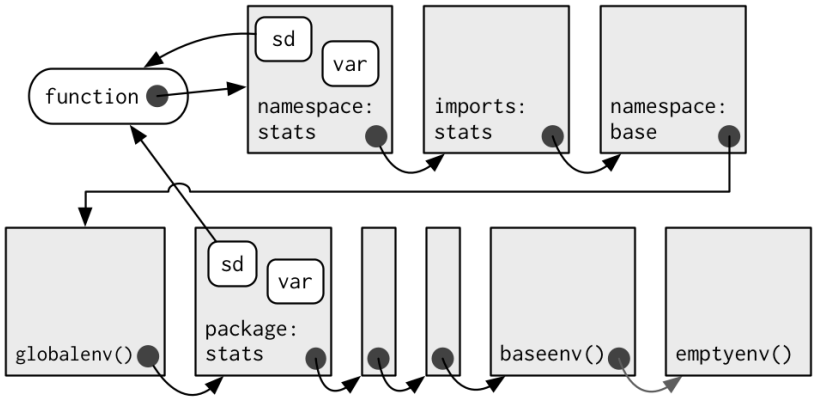
Namespaces are implemented using environments, taking advantage of the fact that functions don't have to live in their enclosing environments. For example, take the base function `sd()`. It's binding and enclosing environments are different:

```
environment(sd)
#> <environment: namespace:stats>
where("sd")
#> <environment: package:stats>
```

The definition of `sd()` uses `var()`, but if we make our own version of `var()` it doesn't affect `sd()`:

```
x <- 1:10
sd(x)
#> [1] 3.03
var <- function(x, na.rm = TRUE) 100
sd(x)
#> [1] 3.03
```

This works because every package has two environments associated with it: the *package* environment and the *namespace* environment. The package environment contains every publicly accessible function, and is placed on the search path. The namespace environment contains all functions (including internal functions), and its parent environment is a special imports environment that contains bindings to all the functions that the package needs. Every exported function in a package is bound into the *package* environment, but enclosed by the *namespace* environment. This complicated relationship is illustrated by the following diagram:



When we type `var` into the console, it's found first in the global environment. When `sd()` looks for `var()` it finds it first in its namespace environment so never looks in the `globalenv()`.

8.3.3 Execution environments

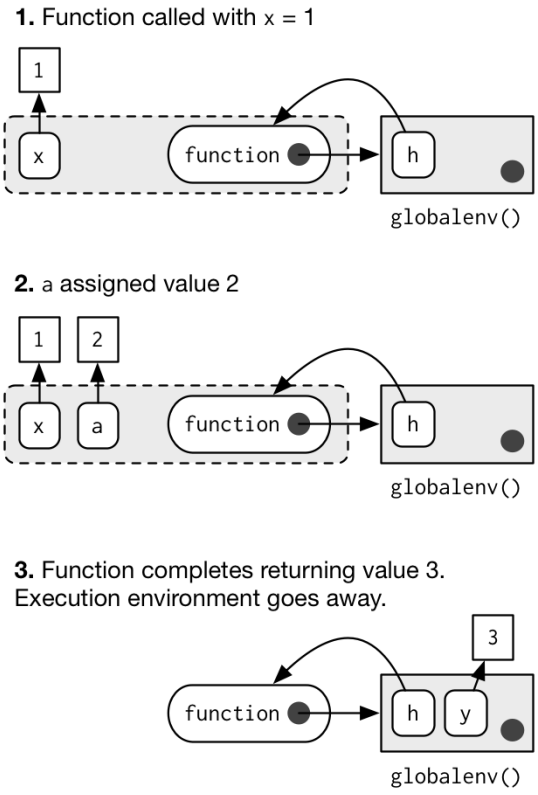
What will the following function return the first time it's run? What about the second?

```
g <- function(x) {
  if (!exists("a", inherits = FALSE)) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
g(10)
g(10)
```

This function returns the same value every time it is called because of the fresh start principle, described in Section 6.2.3. Each time a function is called, a new environment is created to host execution. The parent of the execution environment is the enclosing environment of the function. Once the function has completed, this environment is thrown away.

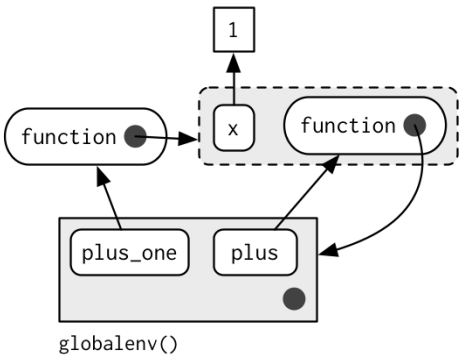
Let's depict that graphically with a simpler function. I draw execution environments around the function they belong to with a dotted border.

```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```



When you create a function inside another function, the enclosing environment of the child function is the execution environment of the parent, and the execution environment is no longer ephemeral. The following example illustrates that idea with a function factory, `plus()`. We use that factory to create a function called `plus_one()`. The enclosing environment of `plus_one()` is the execution environment of `plus()` where `x` is bound to the value 1.

```
plus <- function(x) {
  function(y) x + y
}
plus_one <- plus(1)
identical(parent.env(environment(plus_one)), environment(plus))
#> [1] TRUE
```



You'll learn more about function factories in Chapter 10.

8.3.4 Calling environments

Look at the following code. What do you expect `i()` to return when the code is run?

```
h <- function() {
  x <- 10
  function() {
    x
  }
}
i <- h()
x <- 20
i()
```

The top-level `x` (bound to 20) is a red herring: using the regular scoping rules, `h()` looks first where it is defined and finds that the value associated with `x` is 10. However, it's still meaningful to ask what value `x` is associated within the environment where `i()` is called: `x` is 10 in the

environment where `h()` is defined, but it is 20 in the environment where `h()` is called.

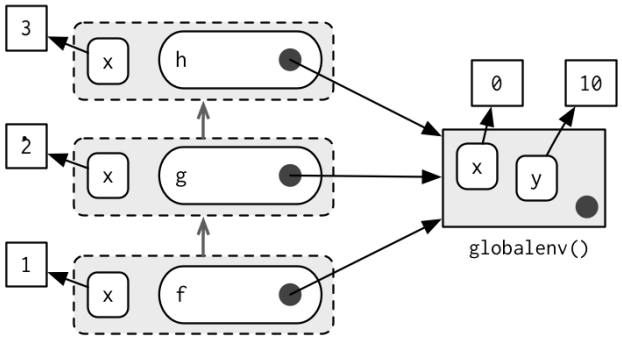
We can access this environment using the unfortunately named `parent.frame()`. This function returns the **environment** where the function was called. We can also use this function to look up the value of names in that environment:

```
f2 <- function() {
  x <- 10
  function() {
    def <- get("x", environment())
    cll <- get("x", parent.frame())
    list(defined = def, called = cll)
  }
}
g2 <- f2()
x <- 20
str(g2())
#> List of 2
#>  $ defined: num 10
#>  $ called : num 20
```

In more complicated scenarios, there's not just one parent call, but a sequence of calls which lead all the way back to the initiating function, called from the top-level. The following code generates a call stack three levels deep. The open-ended arrows represent the calling environment of each execution environment.

```
x <- 0
y <- 10
f <- function() {
  x <- 1
  g()
}
g <- function() {
  x <- 2
  h()
}
h <- function() {
  x <- 3
  x + y
}
```

```
f()
#> [1] 13
```



Note that each execution environment has two parents: a calling environment and an enclosing environment. R's regular scoping rules only use the enclosing parent; `parent.frame()` allows you to access the calling parent.

Looking up variables in the calling environment rather than in the enclosing environment is called **dynamic scoping**. Few languages implement dynamic scoping (Emacs Lisp is a notable exception (`http://www.gnu.org/software/emacs/emacs-paper.html#SEC15`).) This is because dynamic scoping makes it much harder to reason about how a function operates: not only do you need to know how it was defined, you also need to know in what context it was called. Dynamic scoping is primarily useful for developing functions that aid interactive data analysis. It is one of the topics discussed in Chapter 13.

8.3.5 Exercises

1. List the four environments associated with a function. What does each one do? Why is the distinction between enclosing and binding environments particularly important?
2. Draw a diagram that shows the enclosing environments of this function:

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
```

```
    f3(3)
  }
  f2(2)
}
f1(1)
```

3. Expand your previous diagram to show function bindings.
4. Expand it again to show the execution and calling environments.
5. Write an enhanced version of `str()` that provides more information about functions. Show where the function was found and what environment it was defined in.

8.4 Binding names to values

Assignment is the act of binding (or rebinding) a name to a value in an environment. It is the counterpart to scoping, the set of rules that determines how to find the value associated with a name. Compared to most languages, R has extremely flexible tools for binding names to values. In fact, you can not only bind values to names, but you can also bind expressions (promises) or even functions, so that every time you access the value associated with a name, you get something different!

You've probably used regular assignment in R thousands of times. Regular assignment creates a binding between a name and an object in the current environment. Names usually consist of letters, digits, . and _, and can't begin with _. If you try to use a name that doesn't follow these rules, you get an error:

```
_abc <- 1
# Error: unexpected input in "_"
```

Reserved words (like `TRUE`, `NULL`, `if`, and `function`) follow the rules but are reserved by R for other purposes:

```
if <- 10
#> Error: unexpected assignment in "if <-"
```

A complete list of reserved words can be found in `?Reserved`.

It's possible to override the usual rules and use a name with any sequence of characters by surrounding the name with backticks:

```
`a + b` <- 3
`:)` <- "smile"
`    ` <- "spaces"
ls()
#  [1] "    "   ":)"     "a + b"
`:)`
#  [1] "smile"
```

Quotes

You can also create non-syntactic bindings using single and double quotes instead of backticks, but I don't recommend it. The ability to use strings on the left hand side of the assignment arrow is a historical artefact, used before R supported backticks.

The regular assignment arrow, `<-`, always creates a variable in the current environment. The deep assignment arrow, `<<-`, never creates a variable in the current environment, but instead modifies an existing variable found by walking up the parent environments. You can also do deep binding with `assign()`: `name <<- value` is equivalent to `assign("name", value, inherits = TRUE)`.

```
x <- 0
f <- function() {
  x <<- 1
}
f()
x
#> [1] 1
```

If `<<-` doesn't find an existing variable, it will create one in the global environment. This is usually undesirable, because global variables introduce non-obvious dependencies between functions. `<<-` is most often used in conjunction with a closure, as described in Section 10.3.

There are two other special types of binding, delayed and active:

- Rather than assigning the result of an expression immediately, a **delayed binding** creates and stores a promise to evaluate the expression when needed. We can create delayed bindings with the special assignment operator `%<d-%`, provided by the pryr package.

  ```
  library(pryr)
  system.time(b %<d-% {Sys.sleep(1); 1})
  #>    user  system elapsed
  #>   0.000   0.000   0.001
  system.time(b)
  #>    user  system elapsed
  #>   0.000   0.000   1.001
  ```

  `%<d-%` is a wrapper around the base `delayedAssign()` function, which you may need to use directly if you need more control. Delayed bindings are used to implement `autoload()`, which makes R behave as if the package data is in memory, even though it's only loaded from disk when you ask for it.

- **Active** are not bound to a constant object. Instead, they're re-computed every time they're accessed:

  ```
  x %<a-% runif(1)
  x
  #> [1] 0.0808
  x
  #> [1] 0.834
  rm(x)
  ```

  `%<a-%` is a wrapper for the base function `makeActiveBinding()`. You may want to use this function directly if you want more control. Active bindings are used to implement reference class fields.

8.4.1 Exercises

1. What does this function do? How does it differ from `<<-` and why might you prefer it?

   ```
   rebind <- function(name, value, env = parent.frame()) {
     if (identical(env, emptyenv())) {
       stop("Can't find ", name, call. = FALSE)
     } else if (exists(name, envir = env, inherits = FALSE)) {
       assign(name, value, envir = env)
     } else {
   ```

```
    rebind(name, value, parent.env(env))
  }
}
rebind("a", 10)
#> Error: Can't find a
a <- 5
rebind("a", 10)
a
#> [1] 10
```

2. Create a version of `assign()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as single assignment languages (`http://en.wikipedia.org/wiki/Assignment_(computer_science)#Single_assignment`).

3. Write an assignment function that can do active, delayed, and locked bindings. What might you call it? What arguments should it take? Can you guess which sort of assignment it should do based on the input?

8.5 Explicit environments

As well as powering scoping, environments are also useful data structures in their own right because they have **reference semantics**. Unlike most objects in R, when you modify an environment, it does not make a copy. For example, look at this `modify()` function.

```
modify <- function(x) {
  x$a <- 2
  invisible()
}
```

If you apply it to a list, the original list is not changed because modifying a list actually creates and modifies a copy.

```
x_l <- list()
x_l$a <- 1
modify(x_l)
```

```
x_l$a
#> [1] 1
```

However, if you apply it to an environment, the original environment *is* modified:

```
x_e <- new.env()
x_e$a <- 1
modify(x_e)
x_e$a
#> [1] 2
```

Just as you can use a list to pass data between functions, you can also use an environment. When creating your own environment, note that you should set its parent environment to be the empty environment. This ensures you don't accidentally inherit objects from somewhere else:

```
x <- 1
e1 <- new.env()
get("x", envir = e1)
#> [1] 1

e2 <- new.env(parent = emptyenv())
get("x", envir = e2)
#> Error in get("x", envir = e2): object 'x' not found
```

Environments are data structures useful for solving three common problems:

- Avoiding copies of large data.
- Managing state within a package.
- Efficiently looking up values from names.

These are described in turn below.

8.5.1 Avoiding copies

Since environments have reference semantics, you'll never accidentally create a copy. This makes it a useful vessel for large objects. It's a common technique for bioconductor packages which often have to manage

large genomic objects. Changes to R 3.1.0 have made this use substantially less important because modifying a list no longer makes a deep copy. Previously, modifying a single element of a list would cause every element to be copied, an expensive operation if some elements are large. Now, modifying a list efficiently reuses existing vectors, saving much time.

8.5.2 Package state

Explicit environments are useful in packages because they allow you to maintain state across function calls. Normally, objects in a package are locked, so you can't modify them directly. Instead, you can do something like this:

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1

get_a <- function() {
  my_env$a
}
set_a <- function(value) {
  old <- my_env$a
  my_env$a <- value
  invisible(old)
}
```

Returning the old value from setter functions is a good pattern because it makes it easier to reset the previous value in conjunction with `on.exit()` (see more in Section 6.6.1).

8.5.3 As a hashmap

A hashmap is a data structure that takes constant, O(1), time to find an object based on its name. Environments provide this behaviour by default, so can be used to simulate a hashmap. See the CRAN package `hash` for a complete development of this idea.

8.6 Quiz answers

1. There are four ways: every object in an environment must have a name; order doesn't matter; environments have parents; environments have reference semantics.
2. The parent of the global environment is the last package that you loaded. The only environment that doesn't have a parent is the empty environment.
3. The enclosing environment of a function is the environment where it was created. It determines where a function looks for variables.
4. Use `parent.frame()`.
5. `<-` always creates a binding in the current environment; `<<-` rebinds an existing name in a parent of the current environment.

9

Debugging, condition handling, and defensive programming

What happens when something goes wrong with your R code? What do you do? What tools do you have to address the problem? This chapter will teach you how to fix unanticipated problems (debugging), show you how functions can communicate problems and how you can take action based on those communications (condition handling), and teach you how to avoid common problems before they occur (defensive programming).

Debugging is the art and science of fixing unexpected problems in your code. In this section you'll learn the tools and techniques that help you get to the root cause of an error. You'll learn general strategies for debugging, useful R functions like `traceback()` and `browser()`, and interactive tools in RStudio.

Not all problems are unexpected. When writing a function, you can often anticipate potential problems (like a non-existent file or the wrong type of input). Communicating these problems to the user is the job of **conditions**: errors, warnings, and messages.

- Fatal errors are raised by `stop()` and force all execution to terminate. Errors are used when there is no way for a function to continue.
- Warnings are generated by `warning()` and are used to display potential problems, such as when some elements of a vectorised input are invalid, like `log(-1:2)`.
- Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user (`?suppressMessages()`). I often use messages to let the user know what value the function has chosen for an important missing argument.

Conditions are usually displayed prominently, in a bold font or coloured red depending on your R interface. You can tell them apart because errors always start with "Error" and warnings with "Warning message".

Function authors can also communicate with their users with `print()` or `cat()`, but I think that's a bad idea because it's hard to capture and selectively ignore this sort of output. Printed output is not a condition, so you can't use any of the useful condition handling tools you'll learn about below.

Condition handling tools, like `withCallingHandlers()`, `tryCatch()`, and `try()` allow you to take specific actions when a condition occurs. For example, if you're fitting many models, you might want to continue fitting the others even if one fails to converge. R offers an exceptionally powerful condition handling system based on ideas from Common Lisp, but it's currently not very well documented or often used. This chapter will introduce you to the most important basics, but if you want to learn more, I recommend the following two sources:

- *A prototype of a condition system for R* (`http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html`) by Robert Gentleman and Luke Tierney. This describes an early version of R's condition system. While the implementation has changed somewhat since this document was written, it provides a good overview of how the pieces fit together, and some motivation for its design.
- *Beyond Exception Handling: Conditions and Restarts* (`http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html`) by Peter Seibel. This describes exception handling in Lisp, which happens to be very similar to R's approach. It provides useful motivation and more sophisticated examples. I have provided an R translation of the chapter at `http://adv-r.had.co.nz/beyond-exception-handling.html`.

The chapter concludes with a discussion of "defensive" programming: ways to avoid common errors before they occur. In the short run you'll spend more time writing code, but in the long run you'll save time because error messages will be more informative and will let you narrow in on the root cause more quickly. The basic principle of defensive programming is to "fail fast", to raise an error as soon as something goes wrong. In R, this takes three particular forms: checking that inputs are correct, avoiding non-standard evaluation, and avoiding functions that can return different types of output.

Quiz

Want to skip this chapter? Go for it, if you can answer the questions below. Find the answers at the end of the chapter in Section 9.5.

1. How can you find out where an error occured?
2. What does `browser()` do? List the five useful single-key commands that you can use inside of a `browser()` environment.
3. What function do you use to ignore errors in block of code?
4. Why might you want to create an error with a custom S3 class?

Outline

1. Section 9.1 outlines a general approach for finding and resolving bugs.
2. Section 9.2 introduces you to the R functions and Rstudio features that help you locate exactly where an error occurred.
3. Section 9.3 shows you how you can catch conditions (errors, warnings, and messages) in your own code. This allows you to create code that's both more robust and more informative in the presence of errors.
4. Section 9.4 introduces you to some important techniques for defensive programming, techniques that help prevent bugs from occurring in the first place.

9.1 Debugging techniques

> "Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true."
>
> —Norm Matloff

Debugging code is challenging. Many bugs are subtle and hard to find. Indeed, if a bug was obvious, you probably would've been able to avoid it in the first place. While it's true that with a good technique, you can productively debug a problem with just `print()`, there are times when additional help would be welcome. In this section, we'll discuss some useful tools, which R and RStudio provide, and outline a general procedure for debugging.

While the procedure below is by no means foolproof, it will hopefully help you to organise your thoughts when debugging. There are four steps:

1. **Realise that you have a bug**

   If you're reading this chapter, you've probably already completed this step. It is a surprisingly important one: you can't fix a bug until you know it exists. This is one reason why automated test suites are important when producing high-quality code. Unfortunately, automated testing is outside the scope of this book, but you can read more about it at `http://r-pkgs.had.co.nz/tests.html`.

2. **Make it repeatable**

   Once you've determined you have a bug, you need to be able to reproduce it on command. Without this, it becomes extremely difficult to isolate its cause and to confirm that you've successfully fixed it.

   Generally, you will start with a big block of code that you know causes the error and then slowly whittle it down to get to the smallest possible snippet that still causes the error. Binary search is particularly useful for this. To do a binary search, you repeatedly remove half of the code until you find the bug. This is fast because, with each step, you reduce the amount of code to look through by half.

   If it takes a long time to generate the bug, it's also worthwhile to figure out how to generate it faster. The quicker you can do this, the quicker you can figure out the cause.

   As you work on creating a minimal example, you'll also discover similar inputs that don't trigger the bug. Make note of them: they will be helpful when diagnosing the cause of the bug.

   If you're using automated testing, this is also a good time to create an automated test case. If your existing test coverage is low, take the opportunity to add some nearby tests to ensure that existing good behaviour is preserved. This reduces the chances of creating a new bug.

3. **Figure out where it is**

   If you're lucky, one of the tools in the following section will help you to quickly identify the line of code that's causing the bug. Usually, however, you'll have to think a bit more about the problem. It's a great idea to adopt the scientific method. Generate hypotheses, design experiments to test them, and record your results. This may seem like a lot of work, but a systematic approach will end up saving you time. I often

waste a lot of time relying on my intuition to solve a bug ("oh, it must be an off-by-one error, so I'll just subtract 1 here"), when I would have been better off taking a systematic approach.

4. **Fix it and test it**

   Once you've found the bug, you need to figure out how to fix it and to check that the fix actually worked. Again, it's very useful to have automated tests in place. Not only does this help to ensure that you've actually fixed the bug, it also helps to ensure you haven't introduced any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.

9.2 Debugging tools

To implement a strategy of debugging, you'll need tools. In this section, you'll learn about the tools provided by R and the RStudio IDE. RStudio's integrated debugging support makes life easier by exposing existing R tools in a user friendly way. I'll show you both the R and RStudio ways so that you can work with whatever environment you use. You may also want to refer to the official RStudio debugging documentation (`http://www.rstudio.com/ide/docs/debugging/overview`) which always reflects the tools in the latest version of RStudio.

There are three key debugging tools:

- RStudio's error inspector and `traceback()` which list the sequence of calls that lead to the error.
- RStudio's "Rerun with Debug" tool and `options(error = browser)` which open an interactive session where the error occurred.
- RStudio's breakpoints and `browser()` which open an interactive session at an arbitrary location in the code.

I'll explain each tool in more detail below.

You shouldn't need to use these tools when writing new functions. If you find yourself using them frequently with new code, you may want

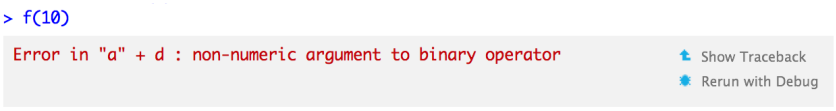
to reconsider your approach. Instead of trying to write one big function all at once, work interactively on small pieces. If you start small, you can quickly identify why something doesn't work. But if you start large, you may end up struggling to identify the source of the problem.

9.2.1 Determining the sequence of calls

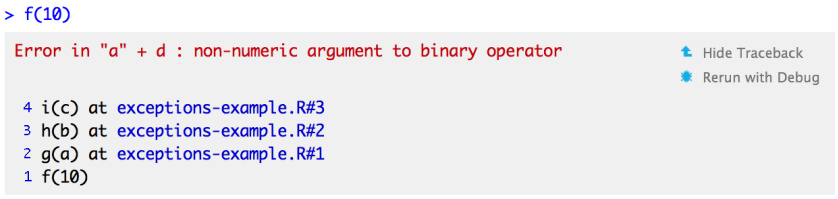
The first tool is the **call stack**, the sequence of calls that lead up to an error. Here's a simple example: you can see that `f()` calls `g()` calls `h()` calls `i()` which adds together a number and a string creating a error:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

When we run this code in Rstudio we see:



Two options appear to the right of the error message: "Show Traceback" and "Rerun with Debug". If you click "Show traceback" you see:



If you're not using Rstudio, you can use `traceback()` to get the same information:

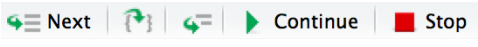
```
traceback()
# 4: i(c) at exceptions-example.R#3
# 3: h(b) at exceptions-example.R#2
# 2: g(a) at exceptions-example.R#1
# 1: f(10)
```

Read the call stack from bottom to top: the initial call is `f()`, which calls `g()`, then `h()`, then `i()`, which triggers the error. If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`. These are clickable in Rstudio, and will take you to the corresponding line of code in the editor.

Sometimes this is enough information to let you track down the error and fix it. However, it's usually not. `traceback()` shows you where the error occurred, but not why. The next useful tool is the interactive debugger, which allows you to pause execution of a function and interactively explore its state.

9.2.2 Browsing on error

The easiest way to enter the interactive debugger is through RStudio's "Rerun with Debug" tool. This reruns the command that created the error, pausing execution where the error occurred. You're now in an interactive state inside the function, and you can interact with any object defined there. You'll see the corresponding code in the editor (with the statement that will be run next highlighted), objects in the current environment in the "Environment" pane, the call stack in a "Traceback" pane, and you can run arbitrary R code in the console.

As well as any regular R function, there are a few special commands you can use in debug mode. You can access them either with the Rstudio toolbar (

) or with the keyboard:

- Next, `n`: executes the next step in the function. Be careful if you have a variable named `n`; to print it you'll need to do `print(n)`.
- Step into,  or `s`: works like next, but if the next step is a function, it will step into that function so you can work through each line.
- Finish,  or `f`: finishes execution of the current loop or function.
- Continue, `c`: leaves interactive debugging and continues regular execution of the function. This is useful if you've fixed the bad state and want to check that the function proceeds correctly.
- Stop, `Q`: stops debugging, terminates the function, and returns to the global workspace. Use this once you've figured out where the problem is, and you're ready to fix it and reload the code.

There are two other slightly less useful commands that aren't available in the toolbar:

- Enter: repeats the previous command. I find this too easy to activate accidentally, so I turn it off using `options(browserNLdisabled = TRUE)`.
- `where`: prints stack trace of active calls (the interactive equivalent of `traceback`).

To enter this style of debugging outside of RStudio, you can use the `error` option which specifies a function to run when an error occurs. The function most similar to Rstudio's debug is `browser()`: this will start an interactive console in the environment where the error occurred. Use `options(error = browser)` to turn it on, re-run the previous command, then use `options(error = NULL)` to return to the default error behaviour. You could automate this with the `browseOnce()` function as defined below:

```
browseOnce <- function() {
  old <- getOption("error")
  function() {
    options(error = old)
    browser()
  }
}
options(error = browseOnce())

f <- function() stop("!")
# Enters browser
f()
# Runs normally
f()
```

(You'll learn more about functions that return functions in Chapter 10.)

There are two other useful functions that you can use with the `error` option:

- `recover` is a step up from `browser`, as it allows you to enter the environment of any of the calls in the call stack. This is useful because often the root cause of the error is a number of calls back.

- `dump.frames` is an equivalent to `recover` for non-interactive code. It creates a `last.dump.rda` file in the current working directory. Then, in a later interactive R session, you load that file, and use `debugger()` to enter an interactive debugger with the same interface as `recover()`. This allows interactive debugging of batch code.

```
# In batch R process ----
dump_and_quit <- function() {
  # Save debugging info to file last.dump.rda
  dump.frames(to.file = TRUE)
  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

To reset error behaviour to the default, use `options(error = NULL)`. Then errors will print a message and abort function execution.

9.2.3 Browsing arbitrary code

As well as entering an interactive console on error, you can enter it at an arbitrary code location by using either an Rstudio breakpoint or `browser()`. You can set a breakpoint in Rstudio by clicking to the left of the line number, or pressing `Shift + F9`. Equivalently, add `browser()` where you want execution to pause. Breakpoints behave similarly to `browser()` but they are easier to set (one click instead of nine key presses), and you don't run the risk of accidentally including a `browser()` statement in your source code. There are two small downsides to breakpoints:

- There are a few unusual situations in which breakpoints will not work: read breakpoint troubleshooting (`http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting`) for more details.
- RStudio currently does not support conditional breakpoints, whereas you can always put `browser()` inside an `if` statement.

As well as adding `browser()` yourself, there are two other functions that will add it to code:

- `debug()` inserts a browser statement in the first line of the specified function. `undebug()` removes it. Alternatively, you can use `debugonce()` to browse only on the next run.

- `utils::setBreakpoint()` works similarly, but instead of taking a function name, it takes a file name and line number and finds the appropriate function for you.

These two functions are both special cases of `trace()`, which inserts arbitrary code at any position in an existing function. `trace()` is occasionally useful when you're debugging code that you don't have the source for. To remove tracing from a function, use `untrace()`. You can only perform one trace per function, but that one trace can call multiple functions.

9.2.4 The call stack: `traceback()`, `where`, and `recover()`

Unfortunately the call stacks printed by `traceback()`, `browser()` + `where`, and `recover()` are not consistent. The following table shows how the call stacks from a simple nested set of calls are displayed by the three tools.

`traceback()`	`where`	`recover()`
`4: stop("Error")`	`where 1: stop("Error")`	`1: f()`
`3: h(x)`	`where 2: h(x)`	`2: g(x)`
`2: g(x)`	`where 3: g(x)`	`3: h(x)`
`1: f()`	`where 4: f()`	

Note that numbering is different between `traceback()` and `where`, and that `recover()` displays calls in the opposite order, and omits the call to `stop()`. RStudio displays calls in the same order as `traceback()` but omits the numbers.

9.2.5 Other types of failure

There are other ways for a function to fail apart from throwing an error or returning an incorrect result.

- A function may generate an unexpected warning. The easiest way to track down warnings is to convert them into errors with `options(warn`

`= 2`) and use the regular debugging tools. When you do this you'll see some extra calls in the call stack, like `doWithOneRestart()`, `withOneRestart()`, `withRestarts()`, and `.signalSimpleWarning()`. Ignore these: they are internal functions used to turn warnings into errors.

- A function may generate an unexpected message. There's no built-in tool to help solve this problem, but it's possible to create one:

```
message2error <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}

f <- function() g()
g <- function() message("Hi!")
g()
# Error in message("Hi!"): Hi!
message2error(g())
traceback()
# 10: stop(e) at #2
# 9: (function (e) stop(e))(list(message = "Hi!\n",
#      call = message("Hi!")))
# 8: signalCondition(cond)
# 7: doWithOneRestart(return(expr), restart)
# 6: withOneRestart(expr, restarts[[1L]])
# 5: withRestarts()
# 4: message("Hi!") at #1
# 3: g()
# 2: withCallingHandlers(code, message = function(e) stop(e))
#      at #2
# 1: message2error(g())
```

  As with warnings, you'll need to ignore some of the calls on the traceback (i.e., the first two and the last seven).

- A function might never return. This is particularly hard to debug automatically, but sometimes terminating the function and looking at the call stack is informative. Otherwise, use the basic debugging strategies described above.

- The worst scenario is that your code might crash R completely, leaving you with no way to interactively debug your code. This indicates a bug in the underlying C code. This is hard to debug. Sometimes an interactive debugger, like `gdb`, can be useful, but describing how to use it is beyond the scope of this book.

If the crash is caused by base R code, post a reproducible example to R-help. If it's in a package, contact the package maintainer. If it's your own C or C++ code, you'll need to use numerous `print()` statements to narrow down the location of the bug, and then you'll need to use many more print statements to figure out which data structure doesn't have the properties that you expect.

9.3 Condition handling

Unexpected errors require interactive debugging to figure out what went wrong. Some errors, however, are expected, and you want to handle them automatically. In R, expected errors crop up most frequently when you're fitting many models to different datasets, such as bootstrap replicates. Sometimes the model might fail to fit and throw an error, but you don't want to stop everything. Instead, you want to fit as many models as possible and then perform diagnostics after the fact.

In R, there are three tools for handling conditions (including errors) programmatically:

- `try()` gives you the ability to continue execution even when an error occurs.
- `tryCatch()` lets you specify **handler** functions that control what happens when a condition is signalled.
- `withCallingHandlers()` is a variant of `tryCatch()` that runs its handlers in a different context. It's rarely needed, but is useful to be aware of.

The following sections describe these tools in more detail.

9.3.1 Ignore errors with try

`try()` allows execution to continue even after an error has occurred. For example, normally if you run a function that throws an error, it terminates immediately and doesn't return a value:

```
f1 <- function(x) {
  log(x)
```

```
  10
}
f1("x")
#> Error in log(x): non-numeric argument to mathematical function
```

However, if you wrap the statement that creates the error in `try()`, the error message will be printed but execution will continue:

```
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
#> Error in log(x) : non-numeric argument to mathematical function
#> [1] 10
```

You can suppress the message with `try(..., silent = TRUE)`.

To pass larger blocks of code to `try()`, wrap them in `{}`:

```
try({
  a <- 1
  b <- "x"
  a + b
})
```

You can also capture the output of the `try()` function. If successful, it will be the last result evaluated in the block (just like a function). If unsuccessful it will be an (invisible) object of class "try-error":

```
success <- try(1 + 2)
failure <- try("a" + "b")
class(success)
#> [1] "numeric"
class(failure)
#> [1] "try-error"
```

`try()` is particularly useful when you're applying a function to multiple elements in a list:

```
elements <- list(1:10, c(-1, 10), c(T, F), letters)
results <- lapply(elements, log)
```

```
#> Warning in lapply(elements, log): NaNs produced
#> Error in FUN(X[[4L]], ...): non-numeric argument to mathematical function
results <- lapply(elements, function(x) try(log(x)))
#> Warning in log(x): NaNs produced
```

There isn't a built-in function to test for the try-error class, so we'll define one. Then you can easily find the locations of errors with `sapply()` (as discussed in Chapter 11), and extract the successes or look at the inputs that lead to failures.

```
is.error <- function(x) inherits(x, "try-error")
succeeded <- !sapply(results, is.error)

# look at successful results
str(results[succeeded])
#> List of 3
#>  $ : num [1:10] 0 0.693 1.099 1.386 1.609 ...
#>  $ : num [1:2] NaN 2.3
#>  $ : num [1:2] 0 -Inf

# look at inputs that failed
str(elements[!succeeded])
#> List of 1
#>  $ : chr [1:26] "a" "b" "c" "d" ...
```

Another useful `try()` idiom is using a default value if an expression fails. Simply assign the default value outside the try block, and then run the risky code:

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

There is also `plyr::failwith()`, which makes this strategy even easier to implement. See Section 12.2 for more details.

9.3.2 Handle conditions with `tryCatch()`

`tryCatch()` is a general tool for handling conditions: in addition to errors, you can take different actions for warnings, messages, and interrupts. You've seen errors (made by `stop()`), warnings (`warning()`) and messages (`message()`) before, but interrupts are new. They can't be generated

directly by the programmer, but are raised when the user attempts to terminate execution by pressing Ctrl + Break, Escape, or Ctrl + C (depending on the platform).

With `tryCatch()` you map conditions to **handlers**, named functions that are called with the condition as an input. If a condition is signalled, `tryCatch()` will call the first handler whose name matches one of the classes of the condition. The only useful built-in names are `error`, `warning`, `message`, `interrupt`, and the catch-all `condition`. A handler function can do anything, but typically it will either return a value or create a more informative error message. For example, the `show_condition()` function below sets up handlers that return the type of condition signalled:

```
show_condition <- function(code) {
  tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
  )
}
show_condition(stop("!"))
#> [1] "error"
show_condition(warning("?!"))
#> [1] "warning"
show_condition(message("?"))
#> [1] "message"

# If no condition is captured, tryCatch returns the
# value of the input
show_condition(10)
#> [1] 10
```

You can use `tryCatch()` to implement `try()`. A simple implementation is shown below. `base::try()` is more complicated in order to make the error message look more like what you'd see if `tryCatch()` wasn't used. Note the use of `conditionMessage()` to extract the message associated with the original error.

```
try2 <- function(code, silent = FALSE) {
  tryCatch(code, error = function(c) {
    msg <- conditionMessage(c)
    if (!silent) message(c)
```

```
    invisible(structure(msg, class = "try-error"))
  })
}

try2(1)
#> [1] 1
try2(stop("Hi"))
try2(stop("Hi"), silent = TRUE)
```

As well as returning default values when a condition is signalled, handlers can be used to make more informative error messages. For example, by modifying the message stored in the error condition object, the following function wraps `read.csv()` to add the file name to any errors:

```
read.csv2 <- function(file, ...) {
  tryCatch(read.csv(file, ...), error = function(c) {
    c$message <- paste0(c$message, " (in ", file, ")")
    stop(c)
  })
}
read.csv("code/dummy.csv")
#> Error in file(file, "rt"): cannot open the connection
read.csv2("code/dummy.csv")
#> Error in file(file, "rt"): cannot open the connection (in code/dummy.csv)
```

Catching interrupts can be useful if you want to take special action when the user tries to abort running code. But be careful, it's easy to create a loop that you can never escape (unless you kill R)!

```
# Don't let the user interrupt the code
i <- 1
while(i < 3) {
  tryCatch({
    Sys.sleep(0.5)
    message("Try to escape")
  }, interrupt = function(x) {
    message("Try again!")
    i <<- i + 1
  })
}
```

`tryCatch()` has one other argument: `finally`. It specifies a block of

code (not a function) to run regardless of whether the initial expression succeeds or fails. This can be useful for clean up (e.g., deleting files, closing connections). This is functionally equivalent to using `on.exit()` but it can wrap smaller chunks of code than an entire function.

9.3.3 `withCallingHandlers()`

An alternative to `tryCatch()` is `withCallingHandlers()`. There are two main differences between these functions:

- The return value of `tryCatch()` handlers is returned by `tryCatch()`, whereas the return value of `withCallingHandlers()` handlers is ignored:

```
f <- function() stop("!")
tryCatch(f(), error = function(e) 1)
#> [1] 1
withCallingHandlers(f(), error = function(e) 1)
#> Error in f(): !
```

- The handlers in `withCallingHandlers()` are called in the context of the call that generated the condition whereas the handlers in `tryCatch()` are called in the context of `tryCatch()`. This is shown here with `sys.calls()`, which is the run-time equivalent of `traceback()` — it lists all calls leading to the current function.

```
f <- function() g()
g <- function() h()
h <- function() stop("!")

tryCatch(f(), error = function(e) print(sys.calls()))
# [[1]] tryCatch(f(), error = function(e) print(sys.calls()))
# [[2]] tryCatchList(expr, classes, parentenv, handlers)
# [[3]] tryCatchOne(expr, names, parentenv, handlers[[1L]])
# [[4]] value[[3L]](cond)

withCallingHandlers(f(), error = function(e) print(sys.calls()))
# [[1]] withCallingHandlers(f(),
#    error = function(e) print(sys.calls()))
# [[2]] f()
# [[3]] g()
# [[4]] h()
# [[5]] stop("!")
```

```
# [[6]] .handleSimpleError(
#    function (e) print(sys.calls()), "!", quote(h()))
# [[7]] h(simpleError(msg, call))
```

This also affects the order in which `on.exit()` is called.

These subtle differences are rarely useful, except when you're trying to capture exactly what went wrong and pass it on to another function. For most purposes, you should never need to use `withCallingHandlers()`.

9.3.4 Custom signal classes

One of the challenges of error handling in R is that most functions just call `stop()` with a string. That means if you want to figure out if a particular error occurred, you have to look at the text of the error message. This is error prone, not only because the text of the error might change over time, but also because many error messages are translated, so the message might be completely different to what you expect.

R has a little known and little used feature to solve this problem. Conditions are S3 classes, so you can define your own classes if you want to distinguish different types of error. Each condition signalling function, `stop()`, `warning()`, and `message()`, can be given either a list of strings, or a custom S3 condition object. Custom condition objects are not used very often, but are very useful because they make it possible for the user to respond to different errors in different ways. For example, "expected" errors (like a model failing to converge for some input datasets) can be silently ignored, while unexpected errors (like no disk space available) can be propagated to the user.

R doesn't come with a built-in constructor function for conditions, but we can easily add one. Conditions must contain `message` and `call` components, and may contain other useful components. When creating a new condition, it should always inherit from `condition` and one of `error`, `warning`, or `message`.

```
condition <- function(subclass, message, call = sys.call(-1), ...) {
  structure(
    class = c(subclass, "condition"),
    list(message = message, call = call),
    ...
  )
}
is.condition <- function(x) inherits(x, "condition")
```

You can signal an arbitrary condition with `signalCondition()`, but nothing will happen unless you've instantiated a custom signal handler (with `tryCatch()` or `withCallingHandlers()`). Instead, use `stop()`, `warning()`, or `message()` as appropriate to trigger the usual handling. R won't complain if the class of your condition doesn't match the function, but you should avoid this in real code.

```
c <- condition(c("my_error", "error"), "This is an error")
signalCondition(c)
# NULL
stop(c)
# Error: This is an error
warning(c)
# Warning message: This is an error
message(c)
# This is an error
```

You can then use `tryCatch()` to take different actions for different types of errors. In this example we make a convenient `custom_stop()` function that allows us to signal error conditions with arbitrary classes. In a real application, it would be better to have individual S3 constructor functions that you could document, describing the error classes in more detail.

```
custom_stop <- function(subclass, message, call = sys.call(-1),
                        ...) {
  c <- condition(c(subclass, "error"), message, call = call, ...)
  stop(c)
}

my_log <- function(x) {
  if (!is.numeric(x))
    custom_stop("invalid_class", "my_log() needs numeric input")
  if (any(x < 0))
    custom_stop("invalid_value", "my_log() needs positive inputs")

  log(x)
}
tryCatch(
  my_log("a"),
  invalid_class = function(c) "class",
  invalid_value = function(c) "value"
```

```
)
#> [1] "class"
```

Note that when using `tryCatch()` with multiple handlers and custom classes, the first handler to match any class in the signal's class hierarchy is called, not the best match. For this reason, you need to make sure to put the most specific handlers first:

```
tryCatch(customStop("my_error", "!"),
  error = function(c) "error",
  my_error = function(c) "my_error"
)
#> [1] "error"
tryCatch(custom_stop("my_error", "!"),
  my_error = function(c) "my_error",
  error = function(c) "error"
)
#> [1] "my_error"
```

9.3.5 Exercises

- Compare the following two implementations of `message2error()`. What is the main advantage of `withCallingHandlers()` in this scenario? (Hint: look carefully at the traceback.)

  ```
  message2error <- function(code) {
    withCallingHandlers(code, message = function(e) stop(e))
  }
  message2error <- function(code) {
    tryCatch(code, message = function(e) stop(e))
  }
  ```

9.4 Defensive programming

Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to "fail fast": as soon as something wrong is discovered, signal an error. This is more work for the author of the

function (you!), but it makes debugging easier for users because they get errors earlier rather than later, after unexpected input has passed through several functions.

In R, the "fail fast" principle is implemented in three ways:

- Be strict about what you accept. For example, if your function is not vectorised in its inputs, but uses functions that are, make sure to check that the inputs are scalars. You can use `stopifnot()`, the assertthat (`https://github.com/hadley/assertthat`) package, or simple `if` statements and `stop()`.

- Avoid functions that use non-standard evaluation, like `subset`, `transform`, and `with`. These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages. You can learn more about non-standard evaluation in Chapter 13.

- Avoid functions that return different types of output depending on their input. The two biggest offenders are `[` and `sapply()`. Whenever subsetting a data frame in a function, you should always use `drop = FALSE`, otherwise you will accidentally convert 1-column data frames into vectors. Similarly, never use `sapply()` inside a function: always use the stricter `vapply()` which will throw an error if the inputs are incorrect types and return the correct type of output even for zero-length inputs.

There is a tension between interactive analysis and programming. When you're working interactively, you want R to do what you mean. If it guesses wrong, you want to discover that right away so you can fix it. When you're programming, you want functions that signal errors if anything is even slightly wrong or underspecified. Keep this tension in mind when writing functions. If you're writing functions to facilitate interactive data analysis, feel free to guess what the analyst wants and recover from minor misspecifications automatically. If you're writing functions for programming, be strict. Never try to guess what the caller wants.

9.4.1 Exercises

- The goal of the `col_means()` function defined below is to compute the means of all numeric columns in a data frame.

```
col_means <- function(df) {
  numeric <- sapply(df, is.numeric)
  numeric_cols <- df[, numeric]

  data.frame(lapply(numeric_cols, mean))
}
```

However, the function is not robust to unusual inputs. Look at the following results, decide which ones are incorrect, and modify `col_means()` to be more robust. (Hint: there are two function calls in `col_means()` that are particularly prone to problems.)

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
col_means(1:10)
col_means(as.matrix(mtcars))
col_means(as.list(mtcars))

mtcars2 <- mtcars
mtcars2[-1] <- lapply(mtcars2[-1], as.character)
col_means(mtcars2)
```

- The following function "lags" a vector, returning a version of `x` that is `n` values behind the original. Improve the function so that it (1) returns a useful error message if `n` is not a vector, and (2) has reasonable behaviour when `n` is 0 or longer than `x`.

```
lag <- function(x, n = 1L) {
  xlen <- length(x)
  c(rep(NA, n), x[seq_len(xlen - n)])
}
```

9.5 Quiz answers

1. The most useful tool to determine where a error occured is `traceback()`. Or use Rstudio, which displays it automatically where an error occurs.

2. `browser()` pauses execution at the specified line and allows you to enter an interactive environment. In that environment, there are five useful commands: `n`, execute the next command; `s`, step into the next function; `f`, finish the current loop or function; `c`, continue execution normally; `Q`, stop the function and return to the console.
3. You could use `try()` or `tryCatch()`.
4. Because you can then capture specific types of error with `tryCatch()`, rather than relying on the comparison of error strings, which is risky, especially when messages are translated.

Part II

Functional programming

10

Functional programming

R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

The chapter starts by showing a motivating example, removing redundancy and duplication in code used to clean and summarise data. Then you'll learn about the three building blocks of functional programming: anonymous functions, closures (functions written by functions), and lists of functions. These pieces are twined together in the conclusion which shows how to build a suite of tools for numerical integration, starting from very simple primitives. This is a recurring theme in FP: start with small, easy-to-understand building blocks, combine them into more complex structures, and apply them with confidence.

The discussion of functional programming continues in the following two chapters: Chapter 11 explores functions that take functions as arguments and return vectors as output, and Chapter 12 explores functions that take functions as input and return them as output.

Outline

- Section 10.1 motivates functional programming using a common problem: cleaning and summarising data before serious analysis.
- Section 10.2 shows you a side of functions that you might not have known about: you can use functions without giving them a name.
- Section 10.3 introduces the closure, a function written by another function. A closure can access its own arguments, and variables defined in its parent.

- Section 10.4 shows how to put functions in a list, and explains why you might care.
- Section 10.5 concludes the chapter with a case study that uses anonymous functions, closures and lists of functions to build a flexible toolkit for numerical integration.

Prequisites

You should be familiar with the basic rules of lexical scoping, as described in Section 6.2. Make sure you've installed the pryr package with `install.packages("pryr")`

10.1 Motivation

Imagine you've loaded a data file, like the one below, that uses −99 to represent missing values. You want to replace all the −99s with `NA`s.

```
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
#>    a  b c   d   e f
#> 1  1  6 1   5 -99 1
#> 2 10  4 4 -99   9 3
#> 3  7  9 5   4   1 4
#> 4  2  9 3   8   6 8
#> 5  1 10 5   9   8 6
#> 6  6  2 1   3   8 5
```

When you first started writing R code, you might have solved the problem with copy-and-paste:

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
```

```
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

One problem with copy-and-paste is that it's easy to make mistakes. Can you spot the two in the block above? These mistakes are inconsistencies that arose because we didn't have an authorative description of the desired action (replace −99 with `NA`). Duplicating an action makes bugs more likely and makes it harder to change code. For example, if the code for a missing value changes from −99 to 9999, you'd need to make the change in multiple places.

To prevent bugs and to make more flexible code, adopt the "do not repeat yourself", or DRY, principle. Popularised by the "pragmatic programmers" (`http://pragprog.com/about`), Dave Thomas and Andy Hunt, this principle states: "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". FP tools are valuable because they provide tools to reduce duplication.

We can start applying FP ideas by writing a function that fixes the missing values in a single vector:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

This reduces the scope of possible mistakes, but it doesn't eliminate them: you can no longer accidentally type -98 instead of -99, but you can still mess up the name of variable. The next step is to remove this possible source of error by combining two functions. One function, `fix_missing()`, knows how to fix a single vector; the other, `lapply()`, knows how to do something to each column in a data frame.

`lapply()` takes three inputs: `x`, a list; `f`, a function; and `...`, other arguments to pass to `f()`. It applies the function to each element of the list and returns a new list. `lapply(x, f, ...)` is equivalent to the following for loop:

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

The real `lapply()` is rather more complicated since it's implemented in C for efficiency, but the essence of the algorithm is the same. `lapply()` is called a **functional**, because it takes a function as an argument. Functionals are an important part of functional programming. You'll learn more about them in Chapter 11.

We can apply `lapply()` to this problem because data frames are lists. We just need a neat little trick to make sure we get back a data frame, not a list. Instead of assigning the results of `lapply()` to `df`, we'll assign them to `df[]`. R's usual rules ensure that we get a data frame, not a list. (If this comes as a surprise, you might want to read Section 3.3.) Putting these pieces together gives us:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

This code has five advantages over copy and paste:

- It's more compact.
- If the code for a missing value changes, it only needs to be updated in one place.
- It works for any number of columns. There is no way to accidentally miss a column.
- There is no way to accidentally treat one column differently than another.
- It is easy to generalise this technique to a subset of columns:

  ```
  df[1:5] <- lapply(df[1:5], fix_missing)
  ```

The key idea is function composition. Take two simple functions, one which does something to every column and one which fixes missing values, and combine them to fix missing values in every column. Writing

simple functions that can be understood in isolation and then composed is a powerful technique.

What if different columns used different codes for missing values? You might be tempted to copy-and-paste:

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

As before, it's easy to create bugs. Instead we could use closures, functions that make and return functions. Closures allow us to make functions based on a template:

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)

fix_missing_99(c(-99, -999))
#> [1]   NA -999
fix_missing_999(c(-99, -999))
#> [1] -99  NA
```

Extra argument

In this case, you could argue that we should just add another argument:

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

That's a reasonable solution here, but it doesn't always work well in every situation. We'll see more compelling uses for closures in Section 11.5.

Now consider a related problem. Once you've cleaned up your data, you might want to compute the same set of numerical summaries for each variable. You could write code like this:

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

But again, you'd be better off identifying and removing duplicate items. Take a minute or two to think about how you might tackle this problem before reading on.

One approach would be to write a summary function and then apply it to each column:

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

That's a great start, but there's still some duplication. It's easier to see if we make the summary function more realistic:

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}
```

All five functions are called with the same arguments (`x` and `na.rm`) repeated five times. As always, duplication makes our code fragile: it's easier to introduce bugs and harder to adapt to changing requirements.

To remove this source of duplication, you can take advantage of another functional programming technique: storing functions in lists.

```
summary <- function(x) {
  funs <- c(mean, median, sd, mad, IQR)
  lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

This chapter discusses these techniques in more detail. But before you can start learning them, you need to learn the simplest FP tool, the anonymous function.

10.2 Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. Unlike many languages (e.g., C, C++, Python, and Ruby), R doesn't have a special syntax for creating a named function: when you create a function, you use the regular assignment operator to give it a name. If you choose not to give the function a name, you get an **anonymous function**.

You use an anonymous function when it's not worth the effort to give it a name:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Like all functions in R, anonymous functions have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x = 4) g(x) + h(x))
#> $x
#> [1] 4
body(function(x = 4) g(x) + h(x))
#> g(x) + h(x)
environment(function(x = 4) g(x) + h(x))
#> <environment: R_GlobalEnv>
```

You can call an anonymous function without giving it a name, but the code is a little tricky to read because you must use parentheses in two different ways: first, to call a function, and second to make it clear that you want to call the anonymous function itself, as opposed to calling a (possibly invalid) function *inside* the anonymous function:

```
# This does not call the anonymous function.
# (Note that "3" is not a valid function.)
function(x) 3()
#> function(x) 3()

# With appropriate parenthesis, the function is called:
(function(x) 3)()
#> [1] 3

# So this anonymous function syntax
(function(x) x + 3)(10)
#> [1] 13

# behaves exactly the same as
f <- function(x) x + 3
f(10)
#> [1] 13
```

You can call anonymous functions with named arguments, but doing so is a good sign that your function needs a name.

One of the most common uses for anonymous functions is to create

closures, functions made by other functions. Closures are described in the next section.

10.2.1 Exercises

1. Given a function, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?
2. Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in the `mtcars` dataset.
3. Use `integrate()` and an anonymous function to find the area under the curve for the following functions. Use Wolfram Alpha (`http://www.wolframalpha.com/`) to check your answers.
   1. `y = x ^ 2 - x`, x in [0, 10]
   2. `y = sin(x) + cos(x)`, x in [$-\pi$, π]
   3. `y = exp(x) / x`, x in [10, 20]
4. A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use `{}`. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?

10.3 Closures

> "An object is data with functions. A closure is a function with data."
> — John D. Cook

One use of anonymous functions is to create small functions that are not worth naming. Another important use is to create closures, functions written by functions. Closures get their name because they **enclose** the environment of the parent function and can access all its variables. This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

The following example uses this idea to generate a family of power functions in which a parent function (`power()`) creates two child functions (`square()` and `cube()`).

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
#> [1] 4
square(4)
#> [1] 16

cube <- power(3)
cube(2)
#> [1] 8
cube(4)
#> [1] 64
```

When you print a closure, you don't see anything terribly useful:

```
square
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fc4e528d318>
cube
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fc4e4dc6b10>
```

That's because the function itself doesn't change. The difference is the enclosing environment, `environment(square)`. One way to see the contents of the environment is to convert it to a list:

```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

Another way to see what's going on is to use `pryr::unenclose()`. This function replaces variables defined in the enclosing environment with their values:

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```

The parent environment of a closure is the execution environment of the function that created it, as shown by this code:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x7fc4e4e38d88>
environment(zero)
#> <environment: 0x7fc4e4e38d88>
```

The execution environment normally disappears after the function returns a value. However, functions capture their enclosing environments. This means when function a returns function b, function b captures and stores the execution environment of function a, and it doesn't disappear. (This has important consequences for memory use, see **??** for details.)

In R, almost every function is a closure. All functions remember the environment in which they were created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written. The only exception is primitive functions, which call C code directly and don't have an associated environment.

Closures are useful for making function factories, and are one way to manage mutable state in R.

10.3.1 Function factories

A function factory is a factory for making new functions. We've already seen two examples of function factories, `missing_fixer()` and `power()`. You call it with arguments that describe the desired actions, and it returns a function that will do the work for you. For `missing_fixer()` and `power()`, there's not much benefit in using a function factory instead of a single function with multiple arguments. Function factories are most useful when:

- The different levels are more complex, with multiple arguments and complicated bodies.
- Some work only needs to be done once, when the function is generated.

Function factories are particularly well suited to maximum likelihood problems, and you'll see a more compelling use of them in Section 11.5.

10.3.2 Mutable state

Having variables at two levels allows you to maintain state across function invocations. This is possible because while the execution environment is refreshed every time, the enclosing environment is constant. The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name. (Section 8.4 has more details on how it works.)

Together, a static parent environment and `<<-` make it possible to maintain state across function calls. The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initialises the counter `i` in this environment, and then creates a new function.

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

The new function is a closure, and its enclosing environment is the environment created when `new_counter()` is run. Ordinarily, function execution environments are temporary, but a closure maintains access to the environment in which it was created. In the example below, closures `counter_one()` and `counter_two()` each get their own enclosing environments when run, so they can maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```

The counters get around the "fresh start" limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.

What happens if you don't use a closure? What happens if you use `<-` instead of `<<-`? Make predictions about what will happen if you replace `new_counter()` with the variants below, then run the code and check your predictions.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Modifying values in a parent environment is an important technique because it is one way to generate "mutable state" in R. Mutable state is normally hard because every time it looks like you're modifying an

object, you're actually creating and then modifying a copy. However, if you do need mutable objects and your code is not very simple, it's usually better to use reference classes, as described in Section 7.4.

The power of closures is tightly coupled with the more advanced ideas in Chapter 11 and Chapter 12. You'll see many more closures in those two chapters. The following section discusses the third technique of functional programming in R: the ability to store functions in a list.

10.3.3 Exercises

1. Why are functions created by other functions called closures?
2. What does the following statistical function do? What would be a better name for it? (The existing name is a bit of a hint.)

   ```
   bc <- function(lambda) {
     if (lambda == 0) {
       function(x) log(x)
     } else {
       function(x) (x ^ lambda - 1) / lambda
     }
   }
   ```

3. What does `approxfun()` do? What does it return?
4. What does `ecdf()` do? What does it return?
5. Create a function that creates functions that compute the ith central moment (`http://en.wikipedia.org/wiki/Central_moment`) of a numeric vector. You can test it by running the following code:

   ```
   m1 <- moment(1)
   m2 <- moment(2)

   x <- runif(100)
   stopifnot(all.equal(m1(x), 0))
   stopifnot(all.equal(m2(x), var(x) * 99 / 100))
   ```

6. Create a function `pick()` that takes an index, `i`, as an argument and returns a function with an argument `x` that subsets `x` with `i`.

   ```
   lapply(mtcars, pick(5))
   # should do the same as this
   lapply(mtcars, function(x) x[[5]])
   ```

10.4 Lists of functions

In R, functions can be stored in lists. This makes it easier to work with groups of related functions, in the same way a data frame makes it easier to work with groups of related vectors.

We'll start with a simple benchmarking example. Imagine you are comparing the performance of multiple ways of computing the arithmetic mean. You could do this by storing each approach (function) in a list:

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)
```

Calling a function from a list is straightforward. You extract it then call it:

```
x <- runif(1e5)
system.time(compute_mean$base(x))
#>    user  system elapsed
#>   0.001   0.000   0.000
system.time(compute_mean[[2]](x))
#>    user  system elapsed
#>   0.000   0.000   0.001
system.time(compute_mean[["manual"]](x))
#>    user  system elapsed
#>   0.054   0.003   0.057
```

To call each function (e.g., to check that they all return the same results), use `lapply()`. We'll need either an anonymous function or a new named function, since there isn't a built-in function to handle this situation.

```
lapply(compute_mean, function(f) f(x))
#> $base
#> [1] 0.499
#>
#> $sum
#> [1] 0.499
#>
#> $manual
#> [1] 0.499

call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
#> $base
#> [1] 0.499
#>
#> $sum
#> [1] 0.499
#>
#> $manual
#> [1] 0.499
```

To time each function, we can combine `lapply()` and `system.time()`:

```
lapply(compute_mean, function(f) system.time(f(x)))
#> $base
#>    user  system elapsed
#>       0       0       0
#>
#> $sum
#>    user  system elapsed
#>   0.001   0.000   0.000
#>
#> $manual
#>    user  system elapsed
#>   0.048   0.001   0.049
```

Another use for a list of functions is to summarise an object in multiple ways. To do that, we could store each summary function in a list, and then run them all with `lapply()`:

```
x <- 1:10
funs <- list(
```

```
  sum = sum,
  mean = mean,
  median = median
)
lapply(funs, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

What if we wanted our summary functions to automatically remove missing values? One approach would be make a list of anonymous functions that call our summary functions with the appropriate arguments:

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
lapply(funs2, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

This, however, leads to a lot of duplication. Apart from a different function name, each function is almost identical. A better approach would be to modify our `lapply()` call to include the extra argument:

```
lapply(funs, function(f) f(x, na.rm = TRUE))
```

10.4.1 Moving lists of functions to the global environment

From time to time you may create a list of functions that you want to be available without having to use a special syntax. For example, imagine

you want to create HTML code by mapping each tag to an R function. The following example uses a function factory to create functions for the tags `<p>` (paragraph), `<b>` (bold), and `<i>` (italics).

```
simple_tag <- function(tag) {
  force(tag)
  function(...) {
    paste0("<", tag, ">", paste0(...), "</", tag, ">")
  }
}
tags <- c("p", "b", "i")
html <- lapply(setNames(tags, tags), simple_tag)
```

I've put the functions in a list because I don't want them to be available all the time. The risk of a conflict between an existing R function and an HTML tag is high. But keeping them in a list makes code more verbose:

```
html$p("This is ", html$b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
```

Depending on how long we want the effect to last, you have three options to eliminate the use of `html$`:

- For a very temporary effect, you can use `with()`:

  ```
  with(html, p("This is ", b("bold"), " text."))
  #> [1] "<p>This is <b>bold</b> text.</p>"
  ```

- For a longer effect, you can `attach()` the functions to the search path, then `detach()` when you're done:

  ```
  attach(html)
  p("This is ", b("bold"), " text.")
  #> [1] "<p>This is <b>bold</b> text.</p>"
  detach(html)
  ```

- Finally, you could copy the functions to the global environment with `list2env()`. You can undo this by deleting the functions after you're done.

  ```
  list2env(html, environment())
  #> <environment: R_GlobalEnv>
  p("This is ", b("bold"), " text.")
  #> [1] "<p>This is <b>bold</b> text.</p>"
  rm(list = names(html), envir = environment())
  ```

I recommend the first option, using `with()`, because it makes it very clear when code is being executed in a special context and what that context is.

10.4.2 Exercises

1. Implement a summary function that works like `base::summary()`, but uses a list of functions. Modify the function so it returns a closure, making it possible to use it as a function factory.
2. Which of the following commands is equivalent to `with(x, f(z))`?
   (a) `x$f(x$z)`.
   (b) `f(x$z)`.
   (c) `x$f(z)`.
   (d) `f(z)`.
   (e) It depends.

10.5 Case study: numerical integration

To conclude this chapter, I'll develop a simple numerical integration tool using first-class functions. Each step in the development of the tool is driven by a desire to reduce duplication and to make the approach more general.

The idea behind numerical integration is simple: find the area under a curve by approximating the curve with simpler components. The two simplest approaches are the **midpoint** and **trapezoid** rules. The midpoint rule approximates a curve with a rectangle. The trapezoid rule uses a trapezoid. Each takes the function we want to integrate, `f`, and a range of values, from `a` to `b`, to integrate over. For this example, I'll try to integrate `sin x` from 0 to π. This is a good choice for testing because it has a simple answer: 2.

```
midpoint <- function(f, a, b) {
  (b - a) * f((a + b) / 2)
}
```

```
trapezoid <- function(f, a, b) {
  (b - a) / 2 * (f(a) + f(b))
}

midpoint(sin, 0, pi)
#> [1] 3.14
trapezoid(sin, 0, pi)
#> [1] 1.92e-16
```

Neither of these functions gives a very good approximation. To make them more accurate using the idea that underlies calculus: we'll break up the range into smaller pieces and integrate each piece using one of the simple rules. This is called **composite integration**. I'll implement it using two new functions:

```
midpoint_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h * f((points[i] + points[i + 1]) / 2)
  }
  area
}

trapezoid_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h / 2 * (f(points[i]) + f(points[i + 1]))
  }
  area
}

midpoint_composite(sin, 0, pi, n = 10)
#> [1] 2.01
midpoint_composite(sin, 0, pi, n = 100)
#> [1] 2
trapezoid_composite(sin, 0, pi, n = 10)
#> [1] 1.98
```

```
trapezoid_composite(sin, 0, pi, n = 100)
#> [1] 2
```

You'll notice that there's a lot of duplication between `midpoint_composite()` and `trapezoid_composite()`. Apart from the internal rule used to integrate over a range, they are basically the same. From these specific functions you can extract a more general composite integration function:

```
composite <- function(f, a, b, n = 10, rule) {
  points <- seq(a, b, length = n + 1)

  area <- 0
  for (i in seq_len(n)) {
    area <- area + rule(f, points[i], points[i + 1])
  }

  area
}

composite(sin, 0, pi, n = 10, rule = midpoint)
#> [1] 2.01
composite(sin, 0, pi, n = 10, rule = trapezoid)
#> [1] 1.98
```

This function takes two functions as arguments: the function to integrate and the integration rule. We can now add even better rules for integrating over smaller ranges:

```
simpson <- function(f, a, b) {
  (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))
}

boole <- function(f, a, b) {
  pos <- function(i) a + i * (b - a) / 4
  fi <- function(i) f(pos(i))

  (b - a) / 90 *
    (7 * fi(0) + 32 * fi(1) + 12 * fi(2) + 32 * fi(3) + 7 * fi(4))
}

composite(sin, 0, pi, n = 10, rule = simpson)
#> [1] 2
```

```
composite(sin, 0, pi, n = 10, rule = boole)
#> [1] 2
```

It turns out that the midpoint, trapezoid, Simpson, and Boole rules are all examples of a more general family called Newton-Cotes rules (`http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas`). (They are polynomials of increasing complexity.) We can use this common structure to write a function that can generate any general Newton-Cotes rule:

```
newton_cotes <- function(coef, open = FALSE) {
  n <- length(coef) + open

  function(f, a, b) {
    pos <- function(i) a + i * (b - a) / n
    points <- pos(seq.int(0, length(coef) - 1))

    (b - a) / sum(coef) * sum(f(points) * coef)
  }
}

boole <- newton_cotes(c(7, 32, 12, 32, 7))
milne <- newton_cotes(c(2, -1, 2), open = TRUE)
composite(sin, 0, pi, n = 10, rule = milne)
#> [1] 1.99
```

Mathematically, the next step in improving numerical integration is to move from a grid of evenly spaced points to a grid where the points are closer together near the end of the range, such as Gaussian quadrature. That's beyond the scope of this case study, but you could implement it with similar techniques.

10.5.1 Exercises

1. Instead of creating individual functions (e.g., `midpoint()`, `trapezoid()`, `simpson()`, etc.), we could store them in a list. If we did that, how would that change the code? Can you create the list of functions from a list of coefficients for the Newton-Cotes formulae?

2. The trade-off between integration rules is that more complex rules are slower to compute, but need fewer pieces. For `sin()`

in the range $[0, \pi]$, determine the number of pieces needed so that each rule will be equally accurate. Illustrate your results with a graph. How do they change for different functions? `sin(1 / x^2)` is particularly challenging.

11

Functionals

> "To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs."
>
> — Bjarne Stroustrup

A higher-order function is a function that takes a function as an input or returns a function as output. We've already seen one type of higher order function: closures, functions returned by another function. The complement to a closure is a **functional**, a function that takes a function as an input and returns a vector as output. Here's a simple functional: it calls the function provided as input with 1000 random uniform numbers.

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.506
randomise(mean)
#> [1] 0.501
randomise(sum)
#> [1] 489
```

The chances are that you've already used a functional: the three most frequently used are `lapply()`, `apply()`, and `tapply()`. All three take a function as input (among other things) and return a vector as output.

A common use of functionals is as an alternative to for loops. For loops have a bad rap in R. They have a reputation for being slow (although that reputation is only partly true, see **??** for more details). But the real downside of for loops is that they're not very expressive. A for loop conveys that it's iterating over something, but doesn't clearly convey a high level goal. Instead of using a for loop, it's better to use a functional. Each functional is tailored for a specific task, so when you recognise the

functional you know immediately why it's being used. Functionals play other roles as well as replacements for for-loops. They are useful for encapsulating common data manipulation tasks like split-apply-combine, for thinking "functionally", and for working with mathematical functions.

Functionals reduce bugs in your code by better communicating intent. Functionals implemented in base R are well tested (i.e., bug-free) and efficient, because they're used by so many people. Many are written in C, and use special tricks to enhance performance. That said, using functionals will not always produce the fastest code. Instead, it helps you clearly communicate and build tools that solve a wide range of problems. It's a mistake to focus on speed until you know it'll be a problem. Once you have clear, correct code you can make it fast using the techniques you'll learn in **??**.

Outline

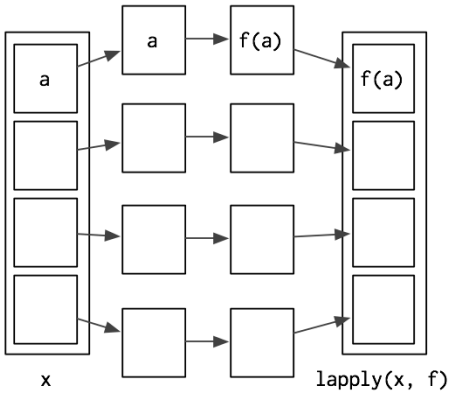
- Section 11.1 introduces your first functional: `lapply()`.
- Section 11.2 shows you variants of `lapply()` that produce different outputs, take different inputs, and distribute computation in different ways.
- Section 11.3 discusses functionals that work with more complex data structures like matrices and arrays.
- Section 11.4 teaches you about the powerful `Reduce()` and `Filter()` functions which are useful for working with lists.
- Section 11.5 discusses functionals that you might be familiar with from mathematics, like root finding, integration, and optimisation.
- Section 11.6 provides some important caveats about when you shouldn't attempt to convert a loop into a functional.
- Section 11.7 finishes off the chapter by showing you how functionals can take a simple building block and use it to create a set of powerful and consistent tools.

Prerequisites

You'll use closures frequently used in conjunction with functionals. If you need a refresher, review Section 10.3.

11.1 My first functional: lapply()

The simplest functional is `lapply()`, which you may already be familiar with. `lapply()` takes a function, applies it to each element in a list, and returns the results in the form of a list. `lapply()` is the building block for many other functionals, so it's important to understand how it works. Here's a pictorial representation:



`lapply()` is written in C for performance, but we can create a simple R implementation that does the same thing:

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

From this code, you can see that `lapply()` is a wrapper for a common for loop pattern: create a container for output, apply `f()` to each component of a list, and fill the container with the results. All other for loop functionals are variations on this theme: they simply use different types of input or output.

`lapply()` makes it easier to work with lists by eliminating much of the boilerplate associated with looping. This allows you to focus on the function that you're applying:

```
# Create some random data
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)

# With a for loop
out <- vector("list", length(l))
for (i in seq_along(l)) {
  out[[i]] <- length(l[[i]])
}
unlist(out)
#>  [1]  3  1  1  2  2 10  5  9  7  2  4 10  8  2  9  7  3  2  2
#> [20]  8

# With lapply
unlist(lapply(l, length))
#>  [1]  3  1  1  2  2 10  5  9  7  2  4 10  8  2  9  7  3  2  2
#> [20]  8
```

(I'm using `unlist()` to convert the output from a list to a vector to make it more compact. We'll see other ways of making the output a vector shortly.)

Since data frames are also lists, `lapply()` is also useful when you want to do something to each column of a data frame:

```
# What class is each column?
unlist(lapply(mtcars, class))
#>       mpg       cyl      disp        hp      drat        wt
#> "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
#>      qsec        vs        am      gear      carb
#> "numeric" "numeric" "numeric" "numeric" "numeric"

# Divide each column by the mean
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

The pieces of `x` are always supplied as the first argument to `f`. If you want to vary a different argument, you can use an anonymous function. The following example varies the amount of trimming applied when computing the mean of a fixed `x`.

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)
unlist(lapply(trims, function(trim) mean(x, trim = trim)))
#> [1] 0.2879 0.0790 0.0535 0.0502
```

11.1.1 Looping patterns

It's useful to remember that there are three basic ways to loop over a vector:

1. loop over the elements: `for (x in xs)`
2. loop over the numeric indices: `for (i in seq_along(xs))`
3. loop over the names: `for (nm in names(xs))`

The first form is usually not a good choice for a for loop because it leads to inefficient ways of saving output. With this form it's very natural to save the output by extending a datastructure, like in this example:

```
xs <- runif(1e3)
res <- c()
for (x in xs) {
  # This is slow!
  res <- c(res, sqrt(x))
}
```

This is slow because each time you extend the vector, R has to copy all of the existing elements. **??** discusses this problem in more depth. Instead, it's much better to create the space you'll need for the output and then fill it in. This is easiest with the second form:

```
res <- numeric(length(xs))
for (i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

Just as there are three basic ways to use a for loop, there are three basic ways to use `lapply()`:

```
lapply(xs, function(x) {})
lapply(seq_along(xs), function(i) {})
lapply(names(xs), function(nm) {})
```

Typically you'd use the first form because `lapply()` takes care of saving the output for you. However, if you need to know the position or name of the element you're working with, you should use the second or third form. Both give you an element's position (`i`, `nm`) and value (`xs[[i]]`, `xs[[nm]]`). If you're struggling to solve a problem using one form, you might find it easier with another.

11.1.2 Exercises

1. Why are the following two invocations of `lapply()` equivalent?

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(100)

lapply(trims, function(trim) mean(x, trim = trim))
lapply(trims, mean, x = x)
```

2. The function below scales a vector so it falls in the range [0, 1]. How would you apply it to every column of a data frame? How would you apply it to every numeric column in a data frame?

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

3. Use both for loops and `lapply()` to fit linear models to the `mtcars` using the formulas stored in this list:

```
formulas <- list(
  mpg ~ disp,
  mpg ~ I(1 / disp),
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)
```

4. Fit the model `mpg ~ disp` to each of the bootstrap replicates of `mtcars` in the list below by using a for loop and `lapply()`. Can you do it without an anonymous function?

```
bootstraps <- lapply(1:10, function(i) {
  rows <- sample(1:nrow(mtcars), rep = TRUE)
  mtcars[rows, ]
})
```

5. For each model in the previous two exercises, extract R^2 using the function below.

```
rsq <- function(mod) summary(mod)$r.squared
```

11.2 For loop functionals: friends of lapply()

The key to using functionals in place of for loops is recognising that common looping patterns are already implemented in existing base functionals. Once you've mastered these existing functionals, the next step is to start writing your own: if you discover you're duplicating the same looping pattern in many places, you should extract it out into its own function.

The following sections build on `lapply()` and discuss:

- `sapply()` and `vapply()`, variants of `lapply()` that produce vectors, matrices, and arrays as **output**, instead of lists.
- `Map()` and `mapply()` which iterate over multiple **input** data structures in parallel.
- `mclapply()` and `mcMap()`, parallel versions of `lapply()` and `Map()`.
- Writing a new function, `rollapply()`, to solve a new problem.

11.2.1 Vector output: `sapply` and `vapply`

`sapply()` and `vapply()` are very similar to `lapply()` except they simplify their output to produce an atomic vector. While `sapply()` guesses, `vapply()` takes an additional argument specifying the output type. `sapply()` is great for interactive use because it saves typing, but if you use it inside your functions you'll get weird errors if you supply the wrong type of input. `vapply()` is more verbose, but gives more informative error messages and never fails silently. It is better suited for use inside other functions.

The following example illustrates these differences. When given a data frame, `sapply()` and `vapply()` return the same results. When given an empty list, `sapply()` returns another empty list instead of the more correct zero-length logical vector.

```
sapply(mtcars, is.numeric)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
vapply(mtcars, is.numeric, logical(1))
```

```
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
sapply(list(), is.numeric)
#> list()
vapply(list(), is.numeric, logical(1))
#> logical(0)
```

If the function returns results of different types or lengths, `sapply()` will silently return a list, while `vapply()` will throw an error. `sapply()` is fine for interactive use because you'll normally notice if something goes wrong, but it's dangerous when writing functions.

The following example illustrates a possible problem when extracting the class of columns in a data frame: if you falsely assume that class only has one value and use `sapply()`, you won't find out about the problem until some future function is given a list instead of a character vector.

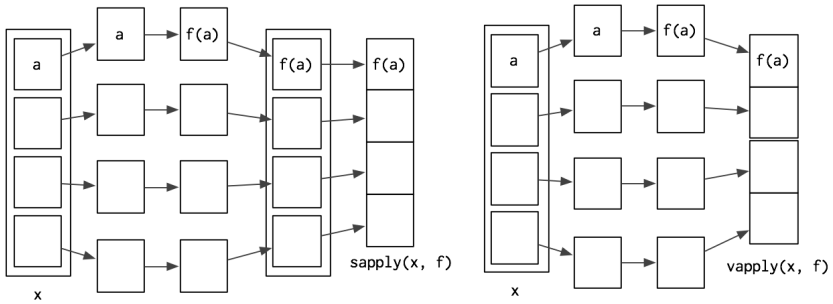
```
df <- data.frame(x = 1:10, y = letters[1:10])
sapply(df, class)
#>         x         y
#> "integer"  "factor"
vapply(df, class, character(1))
#>         x         y
#> "integer"  "factor"

df2 <- data.frame(x = 1:10, y = Sys.time() + 1:10)
sapply(df2, class)
#> $x
#> [1] "integer"
#>
#> $y
#> [1] "POSIXct" "POSIXt"
vapply(df2, class, character(1))
#> Error in vapply(df2, class, character(1)): values must be length 1,
#>  but FUN(X[[2]]) result is length 2
```

`sapply()` is a thin wrapper around `lapply()` that transforms a list into a vector in the final step. `vapply()` is an implementation of `lapply()` that assigns results to a vector (or matrix) of appropriate type instead of as a list. The following code shows a pure R implementation of the essence of `sapply()` and `vapply()` (the real functions have better error handling and preserve names, among other things).

```
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f.value, ...) {
  out <- matrix(rep(f.value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[i], ...)
    stopifnot(
      length(res) == length(f.value),
      typeof(res) == typeof(f.value)
    )
    out[i, ] <- res
  }
  out
}
```



`vapply()` and `sapply()` have different outputs from `lapply()`. The following section discusses `Map()`, which has different inputs.

11.2.2 Multiple inputs: `Map` (and `mapply`)

With `lapply()`, only one argument to the function varies; the others are fixed. This makes it poorly suited for some problems. For example, how would you find a weighted mean when you have two lists, one of observations and the other of weights?

```
# Generate some sample data
xs <- replicate(5, runif(10), simplify = FALSE)
ws <- replicate(5, rpois(10, 5) + 1, simplify = FALSE)
```

It's easy to use `lapply()` to compute the unweighted means:

```
unlist(lapply(xs, mean))
#> [1] 0.678 0.445 0.427 0.469 0.560
```

But how could we supply the weights to `weighted.mean()`? `lapply(x, means, w)` won't work because the additional arguments to `lapply()` are passed to every call. We could change looping forms:

```
unlist(lapply(seq_along(xs), function(i) {
  weighted.mean(xs[[i]], ws[[i]])
}))
#> [1] 0.695 0.464 0.403 0.501 0.521
```

This works, but it's a little clumsy. A cleaner alternative is to use `Map`, a variant of `lapply()`, where all arguments can vary. This lets us write:

```
unlist(Map(weighted.mean, xs, ws))
#> [1] 0.695 0.464 0.403 0.501 0.521
```

Note that the order of arguments is a little different: function is the first argument for `Map()` and the second for `lapply()`.

This is equivalent to:

```
stopifnot(length(xs) == length(ws))
out <- vector("list", length(xs))
for (i in seq_along(xs)) {
  out[[i]] <- weighted.mean(xs[[i]], ws[[i]])
}
```

There's a natural equivalence between `Map()` and `lapply()` because you can always convert a `Map()` to an `lapply()` that iterates over indices. But using `Map()` is more concise, and more clearly indicates what you're trying to do.

`Map` is useful whenever you have two (or more) lists (or data frames) that you need to process in parallel. For example, another way of standardising columns is to first compute the means and then divide by them. We could do this with `lapply()`, but if we do it in two steps, we can more easily check the results at each step, which is particularly important if the first step is more complicated.

```
mtmeans <- lapply(mtcars, mean)
mtmeans[] <- Map(`/`, mtcars, mtmeans)

# In this case, equivalent to
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

If some of the arguments should be fixed and constant, use an anonymous function:

```
Map(function(x, w) weighted.mean(x, w, na.rm = TRUE), xs, ws)
```

We'll see a more compact way to express the same idea in the next chapter.

mapply

You may be more familiar with `mapply()` than `Map()`. I prefer `Map()` because:

- It's equivalent to `mapply` with `simplify = FALSE`, which is almost always what you want.
- Instead of using an anonymous function to provide constant inputs, `mapply` has the `MoreArgs` argument that takes a list of extra arguments that will be supplied, as is, to each call. This breaks R's usual lazy evaluation semantics, and is inconsistent with other functions.

In brief, `mapply()` adds more complication for little gain.

11.2.3 Rolling computations

What if you need a for loop replacement that doesn't exist in base R? You can often create your own by recognising common looping structures and implementing your own wrapper. For example, you might be interested in smoothing your data using a rolling (or running) mean function:

```
rollmean <- function(x, n) {
  out <- rep(NA, length(x))
```

```
  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset + 1)) {
    out[i] <- mean(x[(i - offset):(i + offset - 1)])
  }
  out
}
x <- seq(1, 3, length = 1e2) + runif(1e2)
plot(x)
lines(rollmean(x, 5), col = "blue", lwd = 2)
lines(rollmean(x, 10), col = "red", lwd = 2)
```

But if the noise was more variable (i.e., it has a longer tail), you might worry that your rolling mean was too sensitive to outliers. Instead, you might want to compute a rolling median.

```
x <- seq(1, 3, length = 1e2) + rt(1e2, df = 2) / 3
plot(x)
lines(rollmean(x, 5), col = "red", lwd = 2)
```

To change `rollmean()` to `rollmedian()`, all you need to do is replace `mean` with `median` inside the loop. But instead of copying and pasting to create a new function, we could extract the idea of computing a rolling summary into its own function:

```
rollapply <- function(x, n, f, ...) {
  out <- rep(NA, length(x))

  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset + 1)) {
    out[i] <- f(x[(i - offset):(i + offset)], ...)
  }
  out
}
plot(x)
lines(rollapply(x, 5, median), col = "red", lwd = 2)
```

You might notice that the internal loop looks pretty similar to a `vapply()` loop, so we could rewrite the function as:

```
rollapply <- function(x, n, f, ...) {
  offset <- trunc(n / 2)
  locs <- (offset + 1):(length(x) - n + offset + 1)
  num <- vapply(
    locs,
    function(i) f(x[(i - offset):(i + offset)], ...),
    numeric(1)
  )

  c(rep(NA, offset), num)
}
```

This is effectively the same as the implementation in `zoo::rollapply()`, which provides many more features and much more error checking.

11.2.4 Parallelisation

One interesting thing about the implementation of `lapply()` is that because each iteration is isolated from all others, the order in which they are computed doesn't matter. For example, `lapply3()` scrambles the order of computation, but the results are always the same:

```
lapply3 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in sample(seq_along(x))) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
unlist(lapply(1:10, sqrt))
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
unlist(lapply3(1:10, sqrt))
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

This has a very important consequence: since we can compute each element in any order, it's easy to dispatch the tasks to different cores, and compute them in parallel. This is what `parallel::mclapply()` (and `parallel::mcMap()`) does. (These functions are not available in Windows, but you can use the similar `parLapply()` with a bit more work. See **??** for more details.)

```
library(parallel)
unlist(mclapply(1:10, sqrt, mc.cores = 4))
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

In this case, `mclapply()` is actually slower than `lapply()`. This is because the cost of the individual computations is low, and additional work is needed to send the computation to the different cores and to collect the results.

If we take a more realistic example, generating bootstrap replicates of a linear model for example, the advantages are clearer:

```
boot_df <- function(x) x[sample(nrow(x), rep = T), ]
rsquared <- function(mod) summary(mod)$r.square
boot_lm <- function(i) {
  rsquared(lm(mpg ~ wt + disp, data = boot_df(mtcars)))
}

system.time(lapply(1:500, boot_lm))
#>    user  system elapsed
#>   0.776   0.004   0.786
system.time(mclapply(1:500, boot_lm, mc.cores = 2))
#>    user  system elapsed
#>   0.003   0.008   0.442
```

While increasing the number of cores will not always lead to linear improvement, switching from `lapply()` or `Map()` to its parallelised forms can dramatically improve computational performance.

11.2.5 Exercises

1. Use `vapply()` to:
   a) Compute the standard deviation of every column in a numeric data frame.
   b) Compute the standard deviation of every numeric column in a mixed data frame. (Hint: you'll need to use `vapply()` twice.)
2. Why is using `sapply()` to get the `class()` of each element in a data frame dangerous?
3. The following code simulates the performance of a t-test for non-normal data. Use `sapply()` and an anonymous function to extract the p-value from every trial.

```
trials <- replicate(
  100,
  t.test(rpois(10, 10), rpois(7, 10)),
  simplify = FALSE
)
```

Extra challenge: get rid of the anonymous function by using `[[` directly.

4. What does `replicate()` do? What sort of for loop does it eliminate? Why do its arguments differ from `lapply()` and friends?
5. Implement a version of `lapply()` that supplies `FUN` with both the name and the value of each component.
6. Implement a combination of `Map()` and `vapply()` to create an `lapply()` variant that iterates in parallel over all of its inputs and stores its outputs in a vector (or a matrix). What arguments should the function take?
7. Implement `mcsapply()`, a multicore version of `sapply()`. Can you implement `mcvapply()`, a parallel version of `vapply()`? Why or why not?

11.3 Manipulating matrices and data frames

Functionals can also be used to eliminate loops in common data manipulation tasks. In this section, we'll give a brief overview of the available options, hint at how they can help you, and point you in the right direction to learn more. We'll cover three categories of data structure functionals:

- `apply()`, `sweep()`, and `outer()` work with matrices.
- `tapply()` summarises a vector by groups defined by another vector.
- the `plyr` package, which generalises `tapply()` to make it easy to work with data frames, lists, or arrays as inputs, and data frames, lists, or arrays as outputs.

11.3.1 Matrix and array operations

So far, all the functionals we've seen work with 1d input structures. The three functionals in this section provide useful tools for working with higher-dimensional data structures. `apply()` is a variant of `sapply()` that works with matrices and arrays. You can think of it as an operation that summarises a matrix or array by collapsing each row or column to a single number. It has four arguments:

- `X`, the matrix or array to summarise
- `MARGIN`, an integer vector giving the dimensions to summarise over, 1 = rows, 2 = columns, etc.
- `FUN`, a summary function
- ... other arguments passed on to `FUN`

A typical example of `apply()` looks like this

```
a <- matrix(1:20, nrow = 5)
apply(a, 1, mean)
#> [1]  8.5  9.5 10.5 11.5 12.5
apply(a, 2, mean)
#> [1]  3  8 13 18
```

There are a few caveats to using `apply()`. It doesn't have a simplify argument, so you can never be completely sure what type of output you'll get. This means that `apply()` is not safe to use inside a function unless you carefully check the inputs. `apply()` is also not idempotent in the sense that if the summary function is the identity operator, the output is not always the same as the input:

```
a1 <- apply(a, 1, identity)
identical(a, a1)
#> [1] FALSE
identical(a, t(a1))
#> [1] TRUE
a2 <- apply(a, 2, identity)
identical(a, a2)
#> [1] TRUE
```

(You can put high-dimensional arrays back in the right order using `aperm()`, or use `plyr::aaply()`, which is idempotent.)

`sweep()` allows you to "sweep" out the values of a summary statistic. It is often used with `apply()` to standardise arrays. The following example scales the rows of a matrix so that all values lie between 0 and 1.

```
x <- matrix(rnorm(20, 0, 10), nrow = 4)
x1 <- sweep(x, 1, apply(x, 1, min), `-`)
x2 <- sweep(x1, 1, apply(x1, 1, max), `/`)
```

The final matrix functional is `outer()`. It's a little different in that it takes multiple vector inputs and creates a matrix or array output where the input function is run over every combination of the inputs:

```
# Create a times table
outer(1:3, 1:10, "*")
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
#> [2,]    2    4    6    8   10   12   14   16   18    20
#> [3,]    3    6    9   12   15   18   21   24   27    30
```

Good places to learn more about `apply()` and friends are:

- "Using apply, sapply, lapply in R" (`http://petewerner.blogspot.com/2012/12/using-apply-sapply-lapply-in-r.html`) by Peter Werner.
- "The infamous apply function" (`http://rforpublichealth.blogspot.no/2012/09/the-infamous-apply-function.html`) by Slawa Rokicki.
- "The R apply function - a tutorial with examples" (`http://forgetfulfunctor.blogspot.com/2011/07/r-apply-function-tutorial-with-examples.html`) by axiomOfChoice.
- The stackoverflow question "R Grouping functions: `sapply` vs. `lapply` vs. `apply` vs. `tapply` vs. `by` vs. `aggregate`" (`http://stackoverflow.com/questions/3505701`).

11.3.2 Group apply

You can think about `tapply()` as a generalisation to `apply()` that allows for "ragged" arrays, arrays where each row can have a different number of columns. This is often needed when you're trying to summarise a data set. For example, imagine you've collected pulse rate data from a medical trial, and you want to compare the two groups:

```
pulse <- round(rnorm(22, 70, 10 / 3)) + rep(c(0, 5), c(10, 12))
group <- rep(c("A", "B"), c(10, 12))

tapply(pulse, group, length)
#>  A  B
#> 10 12
tapply(pulse, group, mean)
#>    A    B
#> 70.5 75.0
```

`tapply()` works by creating a "ragged" data structure from a set of inputs, and then applying a function to the individual elements of that structure. The first task is actually what the `split()` function does. It takes two inputs and returns a list which groups elements together from the first vector according to elements, or categories, from the second vector:

```
split(pulse, group)
#> $A
#>  [1] 69 71 74 66 71 67 73 69 73 72
#>
#> $B
#>  [1] 73 79 74 72 74 76 76 68 77 74 79 78
```

Then `tapply()` is just the combination of `split()` and `sapply()`:

```
tapply2 <- function(x, group, f, ..., simplify = TRUE) {
  pieces <- split(x, group)
  sapply(pieces, f, simplify = simplify)
}
tapply2(pulse, group, length)
#>  A  B
#> 10 12
tapply2(pulse, group, mean)
#>    A    B
#> 70.5 75.0
```

Being able to rewrite `tapply()` as a combination of `split()` and `sapply()` is a good indication that we've identified some useful building blocks.

11.3.3 The plyr package

One challenge with using the base functionals is that they have grown organically over time, and have been written by multiple authors. This means that they are not very consistent:

- With `tapply()` and `sapply()`, the simplify argument is called `simplify`. With `mapply()`, it's called `SIMPLIFY`. With `apply()`, the argument is absent.
- `vapply()` is a variant of `sapply()` that allows you to describe what the output should be, but there are no corresponding variants for `tapply()`, `apply()`, or `Map()`.
- The first argument of most base functionals is a vector, but the first argument in `Map()` is a function.

This makes learning these operators challenging, as you have to memorise all of the variations. Additionally, if you think about the possible combinations of input and output types, base R only covers a partial set of cases:

	list	data frame	array
list	`lapply()`		`sapply()`
data frame	`by()`		
array			`apply()`

This was one of the driving motivations behind the creation of the plyr package. It provides consistently named functions with consistently named arguments and covers all combinations of input and output data structures:

	list	data frame	array
list	`llply()`	`ldply()`	`laply()`
data frame	`dlply()`	`ddply()`	`daply()`
array	`alply()`	`adply()`	`aaply()`

Each of these functions splits up the input, applies a function to each piece, and then combines the results. Overall, this process is called "split-apply-combine". You can read more about it and

plyr in "The Split-Apply-Combine Strategy for Data Analysis" (`http://www.jstatsoft.org/v40/i01/`), an open-access article published in the *Journal of Statistical Software*.

11.3.4 Exercises

1. How does `apply()` arrange the output? Read the documentation and perform some experiments.
2. There's no equivalent to `split()` + `vapply()`. Should there be? When would it be useful? Implement one yourself.
3. Implement a pure R version of `split()`. (Hint: use `unique()` and subsetting.) Can you do it without a for loop?
4. What other types of input and output are missing? Brainstorm before you look up some answers in the plyr paper (`http://www.jstatsoft.org/v40/i01/`).

11.4 Manipulating lists

Another way of thinking about functionals is as a set of general tools for altering, subsetting, and collapsing lists. Every functional programming language has three tools for this: `Map()`, `Reduce()`, and `Filter()`. We've seen `Map()` already, and the following sections describe `Reduce()`, a powerful tool for extending two-argument functions, and `Filter()`, a member of an important class of functionals that work with predicates, functions that return a single `TRUE` or `FALSE`.

11.4.1 `Reduce()`

`Reduce()` reduces a vector, `x`, to a single value by recursively calling a function, `f`, two arguments at a time. It combines the first two elements with `f`, then combines the result of that call with the third element, and so on. Calling `Reduce(f, 1:3)` is equivalent to `f(f(1, 2), 3)`. Reduce is also known as fold, because it folds together adjacent elements in the list.

The following two examples show what `Reduce` does with an infix and prefix function:

```
Reduce(`+`, 1:3) # -> ((1 + 2) + 3)
Reduce(sum, 1:3) # -> sum(sum(1, 2), 3)
```

The essence of `Reduce()` can be described by a simple for loop:

```
Reduce2 <- function(f, x) {
  out <- x[[1]]
  for(i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

The real `Reduce()` is more complicated because it includes arguments to control whether the values are reduced from the left or from the right (`right`), an optional initial value (`init`), and an option to output intermediate results (`accumulate`).

`Reduce()` is an elegant way of extending a function that works with two inputs into a function that can deal with any number of inputs. It's useful for implementing many types of recursive operations, like merges and intersections. (We'll see another use in the final case study.) Imagine you have a list of numeric vectors, and you want to find the values that occur in every element:

```
l <- replicate(5, sample(1:10, 15, replace = T), simplify = FALSE)
str(l)
#> List of 5
#>  $ : int [1:15] 10 8 8 1 10 6 6 7 3 9 ...
#>  $ : int [1:15] 5 1 2 10 4 1 1 9 9 6 ...
#>  $ : int [1:15] 1 6 2 7 9 10 8 9 4 6 ...
#>  $ : int [1:15] 9 4 6 10 1 6 9 3 4 4 ...
#>  $ : int [1:15] 4 4 7 7 1 8 1 7 2 3 ...
```

You could do that by intersecting each element in turn:

```
intersect(intersect(intersect(intersect(l[[1]], l[[2]]),
  l[[3]]), l[[4]]), l[[5]])
#> [1] 10  1  6  4  2
```

That's hard to read. With `Reduce()`, the equivalent is:

```
Reduce(intersect, l)
#> [1] 10  1  6  4  2
```

11.4.2 Predicate functionals

A **predicate** is a function that returns a single `TRUE` or `FALSE`, like `is.character`, `all`, or `is.NULL`. A predicate functional applies a predicate to each element of a list or data frame. There are three useful predicate functionals in base R: `Filter()`, `Find()`, and `Position()`.

- `Filter()` selects only those elements which match the predicate.
- `Find()` returns the first element which matches the predicate (or the last element if `right = TRUE`).
- `Position()` returns the position of the first element that matches the predicate (or the last element if `right = TRUE`).

Another useful predicate functional is `where()`, a custom functional that generates a logical vector from a list (or a data frame) and a predicate:

```
where <- function(f, x) {
  vapply(x, f, logical(1))
}
```

The following example shows how you might use these functionals with a data frame:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
where(is.factor, df)
#>     x     y
#> FALSE  TRUE
str(Filter(is.factor, df))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(Find(is.factor, df))
#>  Factor w/ 3 levels "a","b","c": 1 2 3
Position(is.factor, df)
#> [1] 2
```

11.4.3 Exercises

1. Why isn't `is.na()` a predicate function? What base R function is closest to being a predicate version of `is.na()`?

2. Use `Filter()` and `vapply()` to create a function that applies a summary statistic to every numeric column in a data frame.
3. What's the relationship between `which()` and `Position()`? What's the relationship between `where()` and `Filter()`?
4. Implement `Any()`, a function that takes a list and a predicate function, and returns `TRUE` if the predicate function returns `TRUE` for any of the inputs. Implement `All()` similarly.
5. Implement the `span()` function from Haskell: given a list `x` and a predicate function `f`, `span` returns the location of the longest sequential run of elements where the predicate is true. (Hint: you might find `rle()` helpful.)

11.5 Mathematical functionals

Functionals are very common in mathematics. The limit, the maximum, the roots (the set of points where `f(x) = 0`), and the definite integral are all functionals: given a function, they return a single number (or vector of numbers). At first glance, these functions don't seem to fit in with the theme of eliminating loops, but if you dig deeper you'll find out that they are all implemented using an algorithm that involves iteration.

In this section we'll use some of R's built-in mathematical functionals. There are three functionals that work with functions to return single numeric values:

- `integrate()` finds the area under the curve defined by `f()`
- `uniroot()` finds where `f()` hits zero
- `optimise()` finds the location of lowest (or highest) value of `f()`

Let's explore how these are used with a simple function, `sin()`:

```
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
str(uniroot(sin, pi * c(1 / 2, 3 / 2)))
#> List of 5
#>  $ root      : num 3.14
#>  $ f.root    : num 1.22e-16
#>  $ iter      : int 2
```

```
#>  $ init.it   : int NA
#>  $ estim.prec: num 6.1e-05
str(optimise(sin, c(0, 2 * pi)))
#> List of 2
#>  $ minimum  : num 4.71
#>  $ objective: num -1
str(optimise(sin, c(0, pi), maximum = TRUE))
#> List of 2
#>  $ maximum  : num 1.57
#>  $ objective: num 1
```

In statistics, optimisation is often used for maximum likelihood estimation (MLE). In MLE, we have two sets of parameters: the data, which is fixed for a given problem, and the parameters, which vary as we try to find the maximum. These two sets of parameters make the problem well suited for closures. Combining closures with optimisation gives rise to the following approach to solving MLE problems.

The following example shows how we might find the maximum likelihood estimate for λ, if our data come from a Poisson distribution. First, we create a function factory that, given a dataset, returns a function that computes the negative log likelihood (NLL) for parameter `lambda`. In R, it's common to work with the negative since `optimise()` defaults to finding the minimum.

```
poisson_nll <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  function(lambda) {
    n * lambda - sum_x * log(lambda) # + terms not involving lambda
  }
}
```

Note how the closure allows us to precompute values that are constant with respect to the data.

We can use this function factory to generate specific NLL functions for input data. Then `optimise()` allows us to find the best values (the maximum likelihood estimates), given a generous starting range.

```
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
x2 <- c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7, 5, 4, 12, 6, 9)
nll1 <- poisson_nll(x1)
```

```
nll2 <- poisson_nll(x2)

optimise(nll1, c(0, 100))$minimum
#> [1] 32.1
optimise(nll2, c(0, 100))$minimum
#> [1] 5.47
```

We can check that these values are correct by comparing them to the analytic solution: in this case, it's just the mean of the data, 32.1 and 5.467.

Another important mathematical functional is `optim()`. It is a generalisation of `optimise()` that works with more than one dimension. If you're interested in how it works, you might want to explore the `Rvmmin` package, which provides a pure-R implementation of `optim()`. Interestingly `Rvmmin` is no slower than `optim()`, even though it is written in R, not C. For this problem, the bottleneck lies not in controlling the optimisation but with having to evaluate the function multiple times.

11.5.1 Exercises

1. Implement `arg_max()`. It should take a function and a vector of inputs, and return the elements of the input where the function returns the highest value. For example, `arg_max(-10:5, function(x) x ^ 2)` should return -10. `arg_max(-5:5, function(x) x ^ 2)` should return `c(-5, 5)`. Also implement the matching `arg_min()` function.
2. Challenge: read about the fixed point algorithm (`http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3`). Complete the exercises using R.

11.6 Loops that should be left as is

Some loops have no natural functional equivalent. In this section you'll learn about three common cases:

- modifying in place
- recursive functions

- while loops

It's possible to torture these problems to use a functional, but it's not a good idea. You'll create code that is harder to understand, eliminating the main reason for using functionals in the first case.

11.6.1 Modifying in place

If you need to modify part of an existing data frame, it's often better to use a for loop. For example, the following code performs a variable-by-variable transformation by matching the names of a list of functions to the names of variables in a data frame.

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, levels = c("auto", "manual"))
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

We wouldn't normally use `lapply()` to replace this loop directly, but it is *possible*. Just replace the loop with `lapply()` by using `<<-`:

```
lapply(names(trans), function(var) {
  mtcars[[var]] <<- trans[[var]](mtcars[[var]])
})
```

The for loop is gone, but the code is longer and much harder to understand. The reader needs to understand `<<-` and how `x[[y]] <<- z` works (it's not simple!). In short, we've taken a simple, easily understood for loop, and turned it into something few people will understand: not a good idea!

11.6.2 Recursive relationships

It's hard to convert a for loop into a functional when the relationship between elements is not independent, or is defined recursively. For example, exponential smoothing works by taking a weighted average of the current and previous data points. The `exps()` function below implements exponential smoothing with a for loop.

```
exps <- function(x, alpha) {
  s <- numeric(length(x) + 1)
  for (i in seq_along(s)) {
    if (i == 1) {
      s[i] <- x[i]
    } else {
      s[i] <- alpha * x[i - 1] + (1 - alpha) * s[i - 1]
    }
  }
  s
}
x <- runif(6)
exps(x, 0.5)
#> [1] 0.0518 0.0518 0.3078 0.5284 0.6148 0.5978 0.3628
```

We can't eliminate the for loop because none of the functionals we've seen allow the output at position `i` to depend on both the input and output at position `i - 1`.

One way to eliminate the for loop in this case is to solve the recurrence relation (`http://en.wikipedia.org/wiki/Recurrence_relation#Solving`) by removing the recursion and replacing it with explicit references. This requires a new set of mathematical tools, and is challenging, but it can pay off by producing a simpler function.

11.6.3 While loops

Another type of looping construct in R is the `while` loop. It keeps running until some condition is met. `while` loops are more general than `for` loops: you can rewrite every for loop as a while loop, but you can't do the reverse. For example, we could turn this for loop:

```
for (i in 1:10) print(i)
```

into this while loop:

```
i <- 1
while(i <= 10) {
  print(i)
  i <- i + 1
}
```

Not every while loop can be turned into a for loop because many while loops don't know in advance how many times they will be run:

```
i <- 0
while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
}
```

This is a common problem when you're writing simulations.

In this case we can remove the loop by recognising a special feature of the problem. Here we're counting the number of successes before Bernoulli trial with $p = 0.1$ fails. This is a geometric random variable, so you could replace the code with `i <- rgeom(1, 0.1)`. Reformulating the problem in this way is hard to do in general, but you'll benefit greatly if you can do it for your problem.

11.7 A family of functions

To finish off the chapter, this case study shows how you can use functionals to take a simple building block and make it powerful and general. I'll start with a simple idea, adding two numbers together, and use functionals to extend it to summing multiple numbers, computing parallel and cumulative sums, and summing across array dimensions.

We'll start by defining a very simple addition function, one which takes two scalar arguments:

```
add <- function(x, y) {
  stopifnot(length(x) == 1, length(y) == 1,
    is.numeric(x), is.numeric(y))
  x + y
}
```

(We're using R's existing addition operator here, which does much more, but the focus here is on how we can take very simple building blocks and extend them to do more.)

I'll also add an `na.rm` argument. A helper function will make this a bit

easier: if `x` is missing it should return `y`, if `y` is missing it should return `x`, and if both `x` and `y` are missing then it should return another argument to the function: `identity`. This function is probably a bit more general than what we need now, but it's useful if we implement other binary operators.

```
rm_na <- function(x, y, identity) {
  if (is.na(x) && is.na(y)) {
    identity
  } else if (is.na(x)) {
    y
  } else {
    x
  }
}
rm_na(NA, 10, 0)
#> [1] 10
rm_na(10, NA, 0)
#> [1] 10
rm_na(NA, NA, 0)
#> [1] 0
```

This allows us to write a version of `add()` that can deal with missing values if needed:

```
add <- function(x, y, na.rm = FALSE) {
  if (na.rm && (is.na(x) || is.na(y))) rm_na(x, y, 0) else x + y
}
add(10, NA)
#> [1] NA
add(10, NA, na.rm = TRUE)
#> [1] 10
add(NA, NA)
#> [1] NA
add(NA, NA, na.rm = TRUE)
#> [1] 0
```

Why did we pick an identity of `0`? Why should `add(NA, NA, na.rm = TRUE)` return 0? Well, for every other input it returns a number, so even if both arguments are `NA`, it should still do that. What number should it return? We can figure it out because additional is associative, which means that the order of additional doesn't matter. That means that the following two function calls should return the same value:

```
add(add(3, NA, na.rm = TRUE), NA, na.rm = TRUE)
#> [1] 3
add(3, add(NA, NA, na.rm = TRUE), na.rm = TRUE)
#> [1] 3
```

This implies that `add(NA, NA, na.rm = TRUE)` must be 0, and hence `identity = 0` is the correct default.

Now that we have the basics working, we can extend the function to deal with more complicated inputs. One obvious generalisation is to add more than two numbers. We can do this by iteratively adding two numbers: if the input is `c(1, 2, 3)` we compute `add(add(1, 2), 3)`. This is a simple application of `Reduce()`:

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs)
}
r_add(c(1, 4, 10))
#> [1] 15
```

This looks good, but we need to test a few special cases:

```
r_add(NA, na.rm = TRUE)
#> [1] NA
r_add(numeric())
#> NULL
```

These are incorrect. In the first case, we get a missing value even though we've explicitly asked to ignore them. In the second case, we get `NULL` instead of a length one numeric vector (as we do for every other set of inputs).

The two problems are related. If we give `Reduce()` a length one vector, it doesn't have anything to reduce, so it just returns the input. If we give it an input of length zero, it always returns `NULL`. The easiest way to fix this problem is to use the `init` argument of `Reduce()`. This is added to the start of every input vector:

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs, init = 0)
}
r_add(c(1, 4, 10))
```

```
#> [1] 15
r_add(NA, na.rm = TRUE)
#> [1] 0
r_add(numeric())
#> [1] 0
```

`r_add()` is equivalent to `sum()`.

It would be nice to have a vectorised version of `add()` so that we can perform the addition of two vectors of numbers in element-wise fashion. We could use `Map()` or `vapply()` to implement this, but neither is perfect. `Map()` returns a list, instead of a numeric vector, so we need to use `simplify2array()`. `vapply()` returns a vector but it requires us to loop over a set of indices.

```
v_add1 <- function(x, y, na.rm = FALSE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  if (length(x) == 0) return(numeric())
  simplify2array(
    Map(function(x, y) add(x, y, na.rm = na.rm), x, y)
  )
}

v_add2 <- function(x, y, na.rm = FALSE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  vapply(seq_along(x), function(i) add(x[i], y[i], na.rm = na.rm),
    numeric(1))
}
```

A few test cases help to ensure that it behaves as we expect. We're a bit stricter than base R here because we don't do recycling. (You could add that if you wanted, but I find that recycling is a frequent source of silent bugs.)

```
# Both versions give the same results
v_add1(1:10, 1:10)
#>  [1]  2  4  6  8 10 12 14 16 18 20
v_add1(numeric(), numeric())
#> numeric(0)
v_add1(c(1, NA), c(1, NA))
#> [1]  2 NA
v_add1(c(1, NA), c(1, NA), na.rm = TRUE)
#> [1] 2 0
```

Another variant of `add()` is the cumulative sum. We can implement it with `Reduce()` by setting the `accumulate` argument to `TRUE`:

```
c_add <- function(xs, na.rm = FALSE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs,
    accumulate = TRUE)
}
c_add(1:10)
#>  [1]  1  3  6 10 15 21 28 36 45 55
c_add(10:1)
#>  [1] 10 19 27 34 40 45 49 52 54 55
```

This is equivalent to `cumsum()`.

Finally, we might want to define addition for more complicated data structures like matrices. We could create `row` and `col` variants that sum across rows and columns, respectively, or we could go the whole hog and define an array version that could sum across any arbitrary set of dimensions. These are easily implemented as combinations of `add()` and `apply()`.

```
row_sum <- function(x, na.rm = FALSE) {
  apply(x, 1, add, na.rm = na.rm)
}
col_sum <- function(x, na.rm = FALSE) {
  apply(x, 2, add, na.rm = na.rm)
}
arr_sum <- function(x, dim, na.rm = FALSE) {
  apply(x, dim, add, na.rm = na.rm)
}
```

The first two are equivalent to `rowSums()` and `colSums()`.

If every function we have created has an existing equivalent in base R, why did we bother? There are two main reasons:

- Since all variants were implemented by combining a simple binary operator (`add()`) and a well-tested functional (`Reduce()`, `Map()`, `apply()`), we know that our variants will behave consistently.
- We can apply the same infrastructure to other operators, especially those that might not have the full suite of variants in base R.

The downside of this approach is that these implementations are not that efficient. (For example, `colSums(x)` is much faster than `apply(x, 2, sum)`.) However, even if they aren't that fast, simple implementations are still a good starting point because they're less likely to have bugs. When you create faster versions, you can compare the results to make sure your fast versions are still correct.

If you enjoyed this section, you might also enjoy "List out of lambda" (`http://stevelosh.com/blog/2013/03/list-out-of-lambda/`), a blog article by Steve Losh that shows how you can produce high level language structures (like lists) out of more primitive language features (like closures, aka lambdas).

11.7.1 Exercises

1. Implement `smaller` and `larger` functions that, given two inputs, return either the smaller or the larger value. Implement `na.rm = TRUE`: what should the identity be? (Hint: `smaller(x, smaller(NA, NA, na.rm = TRUE), na.rm = TRUE)` must be `x`, so `smaller(NA, NA, na.rm = TRUE)` must be bigger than any other value of x.) Use `smaller` and `larger` to implement equivalents of `min()`, `max()`, `pmin()`, `pmax()`, and new functions `row_min()` and `row_max()`.
2. Create a table that has *and*, *or*, *add*, *multiply*, *smaller*, and *larger* in the columns and *binary operator*, *reducing variant*, *vectorised variant*, and *array variants* in the rows.
    a) Fill in the cells with the names of base R functions that perform each of the roles.
    b) Compare the names and arguments of the existing R functions. How consistent are they? How could you improve them?
    c) Complete the matrix by implementing any missing functions.
3. How does `paste()` fit into this structure? What is the scalar binary function that underlies `paste()`? What are the `sep` and `collapse` arguments to `paste()` equivalent to? Are there any `paste` variants that don't have existing R implementations?

12

Function operators

In this chapter, you'll learn about function operators (FOs). A function operator is a function that takes one (or more) functions as input and returns a function as output. In some ways, function operators are similar to functionals: there's nothing you can't do without them, but they can make your code more readable and expressive, and they can help you write code faster. The main difference is that functionals extract common patterns of loop use, where function operators extract common patterns of anonymous function use.

The following code shows a simple function operator, `chatty()`. It wraps a function, making a new function that prints out its first argument. It's useful because it gives you a window to see how functionals, like `vapply()`, work.

```
chatty <- function(f) {
  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)
chatty(f)(1)
#> Processing 1
#> [1] 1

vapply(s, chatty(f), numeric(1))
#> Processing 3
#> Processing 2
#> Processing 1
#> [1] 9 4 1
```

In the last chapter, we saw that many built-in functionals, like `Reduce()`,

`Filter()`, and `Map()`, have very few arguments, so we had to use anonymous functions to modify how they worked. In this chapter, we'll build specialised substitutes for common anonymous functions that allow us to communicate our intent more clearly. For example, in Section 11.2.2 we used an anonymous function with `Map()` to supply fixed arguments:

```
Map(function(x, y) f(x, y, zs), xs, ys)
```

Later in this chapter, we'll learn about partial application using the `partial()` function. Partial application encapsulates the use of an anonymous function to supply default arguments, and allows us to write succinct code:

```
Map(partial(f, zs = zs), xs, yz)
```

This is an important use of FOs: by transforming the input function, you eliminate parameters from a functional. In fact, as long as the inputs and outputs of the function remain the same, this approach allows your functionals to be more extensible, often in ways you haven't thought of.

The chapter covers four important types of FO: behaviour, input, output, and combining. For each type, I'll show you some useful FOs, and how you can use as another to decompose problems: as combinations of multiple functions instead of combinations of arguments. The goal is not to exhaustively list every possible FO, but to show a selection that demonstrate how they work together with other FP techniques. For your own work, you'll need to think about and experiment with how function operators can help you solve recurring problems.

Outline

- Section 12.1 introduces you to FOs that change the behaviour of a function like automatically logging usage to disk or ensuring that a function is run only once.
- Section 12.2 shows you how to write FOs that manipulate the output of a function. These can do simple things like capturing errors, or fundamentally change what the function does.
- Section 12.3 describes how to modify the inputs to a function using a FO like `Vectorize()` or `partial()`.
- Section 12.4 shows the power of FOs that combine multiple functions with function composition and logical operations.

Prerequisites

As well as writing FOs from scratch, this chapter uses function operators from the memoise, plyr, and pryr packages. Install them by running `install.packages(c("memoise", "plyr", "pryr"))`.

12.1 Behavioural FOs

Behavioural FOs leave the inputs and outputs of a function unchanged, but add some extra behaviour. In this section, we'll look at functions which implement three useful behaviours:

- Add a delay to avoid swamping a server with requests.
- Print to console every n invocations to check on a long running process.
- Cache previous computations to improve performance.

To motivate these behaviours, imagine we want to download a long vector of URLs. That's pretty simple with `lapply()` and `download_file()`:

```
download_file <- function(url, ...) {
  download.file(url, basename(url), ...)
}
lapply(urls, download_file)
```

(`download_file()` is a simple wrapper around `utils::download.file()` which provides a reasonable default for the file name.)

There are a number of useful behaviours we might want to add to this function. If the list was long, we might want to print a . every ten URLs so we know that the function's still working. If we're downloading files over the internet, we might want to add a small delay between each request to avoid hammering the server. Implementing these behaviours in a for loop is rather complicated. We can no longer use `lapply()` because we need an external counter:

```
i <- 1
for(url in urls) {
  i <- i + 1
```

```
  if (i %% 10 == 0) cat(".")
  Sys.delay(1)
  download_file(url)
}
```

Understanding this code is hard because different concerns (iteration, printing, and downloading) are interleaved. In the remainder of this section we'll create FOs that encapsulate each behaviour and allow us to write code like this:

```
lapply(urls, dot_every(10, delay_by(1, download_file)))
```

Implementing `delay_by()` is straightforward, and follows the same basic template that we'll see for the majority of FOs in this chapter:

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}
system.time(runif(100))
#>    user  system elapsed
#>       0       0       0
system.time(delay_by(0.1, runif)(100))
#>    user  system elapsed
#>   0.000   0.000   0.102
```

`dot_every()` is a little bit more complicated because it needs to manage a counter. Fortunately, we saw how to do that in Section 10.3.2.

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")
    i <<- i + 1
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
#> ..........
```

Notice that I've made the function the last argument in each FO. This makes it easier to read when we compose multiple function operators. If the function were the first argument, then instead of:

```
download <- dot_every(10, delay_by(1, download_file))
```

we'd have

```
download <- dot_every(delay_by(download_file, 1), 10)
```

That's harder to follow because (e.g.) the argument of `dot_every()` is far away from its call. This is sometimes called the Dagwood sandwich (`http://en.wikipedia.org/wiki/Dagwood_sandwich`) problem: you have too much filling (too many long arguments) between your slices of bread (parentheses).

I've also tried to give the FOs descriptive names: delay by 1 (second), (print a) dot every 10 (invocations). The more clearly the function names used in your code express your intent, the easier it will be for others (including future you) to read and understand the code.

12.1.1 Memoisation

Another thing you might worry about when downloading multiple files is accidentally downloading the same file multiple times. You could avoid this by calling `unique()` on the list of input URLs, or manually managing a data structure that mapped the URL to the result. An alternative approach is to use memoisation: modify a function to automatically cache its results.

```
library(memoise)

slow_function <- function(x) {
  Sys.sleep(1)
  10
}
system.time(slow_function())
#>    user  system elapsed
#>       0       0       1
system.time(slow_function())
#>    user  system elapsed
```

```
#>       0       0       1
fast_function <- memoise(slow_function)
system.time(fast_function())
#>    user  system elapsed
#>       0       0       1
system.time(fast_function())
#>    user  system elapsed
#>       0       0       0
```

Memoisation is an example of the classic computer science tradeoff of memory versus speed. A memoised function can run much faster because it stores all of the previous inputs and outputs, using more memory.

A realistic use of memoisation is computing the Fibonacci series. The Fibonacci series is defined recursively: the first two values are 1 and 1, then f(n) = f(n - 1) + f(n - 2). A naive version implemented in R would be very slow because, for example, `fib(10)` computes `fib(9)` and `fib(8)`, and `fib(9)` computes `fib(8)` and `fib(7)`, and so on. As a result, the value for each value in the series gets computed many, many times. Memoising `fib()` makes the implementation much faster because each value is computed only once.

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#>    user  system elapsed
#>   0.067   0.002   0.069
system.time(fib(24))
#>    user  system elapsed
#>   0.102   0.000   0.103

fib2 <- memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>    user  system elapsed
#>   0.003   0.000   0.004
system.time(fib2(24))
#>    user  system elapsed
#>   0.001   0.000   0.000
```

It doesn't make sense to memoise all functions. For example, a memoised random number generator is no longer random:

```
runifm <- memoise(runif)
runifm(5)
#> [1] 0.883 0.678 0.073 0.920 0.988
runifm(5)
#> [1] 0.883 0.678 0.073 0.920 0.988
```

Once we understand `memoise()`, it's straightforward to apply to our problem:

```
download <- dot_every(10, memoise(delay_by(1, download_file)))
```

This gives a function that we can easily use with `lapply()`. However, if something goes wrong with the loop inside `lapply()`, it can be difficult to tell what's going on. The next section will show how we can use FOs to pull back the curtain and look inside.

12.1.2 Capturing function invocations

One challenge with functionals is that it can be hard to see what's going on inside of them. It's not easy to pry open their internals like it is with a for loop. Fortunately we can use FOs to peer behind the curtain with `tee()`.

`tee()`, defined below, has three arguments, all functions: `f`, the function to modify; `on_input`, a function that's called with the inputs to `f`; and `on_output`, a function that's called with the output from `f`.

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
  function(...) {
    on_input(...)
    output <- f(...)
    on_output(output)
    output
  }
}
```

(The function is inspired by the unix shell command `tee`, which is used

to split up streams of file operations so that you can both display what's happening and save intermediate results to a file.)

We can use `tee()` to look inside the `uniroot()` functional, and see how it iterates its way to a solution. The following example finds where `x` and `cos(x)` intersect:

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))
show_x <- function(x, ...) cat(sprintf("%+.08f", x), "\n")

# The location where the function is evaluated:
zero <- uniroot(tee(g, on_input = show_x), c(-5, 5))
#> -5.00000000
#> +5.00000000
#> +0.28366219
#> +0.87520341
#> +0.72298040
#> +0.73863091
#> +0.73908529
#> +0.73902425
#> +0.73908529
# The value of the function:
zero <- uniroot(tee(g, on_output = show_x), c(-5, 5))
#> +5.28366219
#> -4.71633781
#> +0.67637474
#> -0.23436269
#> +0.02685676
#> +0.00076012
#> -0.00000026
#> +0.00010189
#> -0.00000026
```

`cat()` allows us to see what's happening as the function runs, but it doesn't give us a way to work with the values after the function as completed. To do that, we could capture the sequence of calls by creating a function, `remember()`, that records every argument called and retrieves them when coerced into a list. The small amount of S3 code needed is explained in Section 7.2.

```
remember <- function() {
  memory <- list()
```

```
  f <- function(...) {
    # This is inefficient!
    memory <<- append(memory, list(...))
    invisible()
  }

  structure(f, class = "remember")
}
as.list.remember <- function(x, ...) {
  environment(x)$memory
}
print.remember <- function(x, ...) {
  cat("Remembering...\n")
  str(as.list(x))
}
```

Now we can draw a picture showing how uniroot zeroes in on the final answer:

```
locs <- remember()
vals <- remember()
zero <- uniroot(tee(g, locs, vals), c(-5, 5))
x <- unlist(as.list(locs))
error <- unlist(as.list(vals))
plot(x, type = "b"); abline(h = 0.739, col = "grey50")
```

```
plot(error, type = "b"); abline(h = 0, col = "grey50")
```

12.1.3 Laziness

The function operators we've seen so far follow a common pattern:

```
funop <- function(f, otherargs) {
  function(...) {
    # maybe do something
    res <- f(...)
```

```
    # maybe do something else
    res
  }
}
```

Unfortunately there's a problem with this implementation because function arguments are lazily evaluated: `f()` may have changed between applying the FO and evaluating the function. This is a particular problem if you're using a for loop or `lapply()` to apply multiple function operators. In the following example, we take a list of functions and delay each one. But when we try to evaluate the mean, we get the sum instead.

```
funs <- list(mean = mean, sum = sum)
funs_m <- lapply(funs, delay_by, delay = 0.1)

funs_m$mean(1:10)
#> [1] 55
```

We can avoid that problem by explicitly forcing the evaluation of `f()`:

```
delay_by <- function(delay, f) {
  force(f)
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}

funs_m <- lapply(funs, delay_by, delay = 0.1)
funs_m$mean(1:10)
#> [1] 5.5
```

It's good practice to do that whenever you create a new FO.

12.1.4 Exercises

1. Write a FO that logs a time stamp and message to a file every time a function is run.
2. What does the following function do? What would be a good name for it?

```
f <- function(g) {
  force(g)
  result <- NULL
  function(...) {
    if (is.null(result)) {
      result <<- g(...)
    }
    result
  }
}
runif2 <- f(runif)
runif2(5)
#> [1] 0.128 0.756 0.459 0.877 0.618
runif2(10)
#> [1] 0.128 0.756 0.459 0.877 0.618
```

3. Modify `delay_by()` so that instead of delaying by a fixed amount of time, it ensures that a certain amount of time has elapsed since the function was last called. That is, if you called `g <- delay_by(1, f); g(); Sys.sleep(2); g()` there shouldn't be an extra delay.

4. Write `wait_until()` which delays execution until a specific time.

5. There are three places we could have added a memoise call: why did we choose the one we did?

   ```
   download <- memoise(dot_every(10, delay_by(1, download_file)))
   download <- dot_every(10, memoise(delay_by(1, download_file)))
   download <- dot_every(10, delay_by(1, memoise(download_file)))
   ```

6. Why is the `remember()` function inefficient? How could you implement it in more efficient way?

7. Why does the following code, from stackoverflow (`http://stackoverflow.com/questions/8440675`), not do what you expect?

   ```
   # return a linear function with slope a and intercept b.
   f <- function(a, b) function(x) a * x + b

   # create a list of functions with different parameters.
   fs <- Map(f, a = c(0, 1), b = c(0, 1))

   fs[[1]](3)
   ```

```
#> [1] 4
# should return 0 * 3 + 0 = 0
```

How can you modify `f` so that it works correctly?

12.2 Output FOs

The next step up in complexity is to modify the output of a function. This could be quite simple, or it could fundamentally change the operation of the function by returning something completely different to its usual output. In this section you'll learn about two simple modifications, `Negate()` and `failwith()`, and two fundamental modifications, `capture_it()` and `time_it()`.

12.2.1 Minor modifications

`base::Negate()` and `plyr::failwith()` offer two minor, but useful, modifications of a function that are particularly handy in conjunction with functionals.

`Negate()` takes a function that returns a logical vector (a predicate function), and returns the negation of that function. This can be a useful shortcut when a function returns the opposite of what you need. The essence of `Negate()` is very simple:

```
Negate <- function(f) {
  force(f)
  function(...) !f(...)
}
(Negate(is.null))(NULL)
#> [1] FALSE
```

I often use this idea to make a function, `compact()`, that removes all null elements from a list:

```
compact <- function(x) Filter(Negate(is.null), x)
```

`plyr::failwith()` turns a function that throws an error into a function

that returns a default value when there's an error. Again, the essence of `failwith()` is simple; it's just a wrapper around `try()`, the function that captures errors and allows execution to continue.

```
failwith <- function(default = NULL, f, quiet = FALSE) {
  force(f)
  function(...) {
    out <- default
    try(out <- f(...), silent = quiet)
    out
  }
}
log("a")
#> Error in log("a"): non-numeric argument to mathematical function
failwith(NA, log)("a")
#> [1] NA
failwith(NA, log, quiet = TRUE)("a")
#> [1] NA
```

(If you haven't seen `try()` before, it's discussed in more detail in Section 9.3.1.)

`failwith()` is very useful in conjunction with functionals: instead of the failure propagating and terminating the higher-level loop, you can complete the iteration and then find out what went wrong. For example, imagine you're fitting a set of generalised linear models (GLMs) to a list of data frames. While GLMs can sometimes fail because of optimisation problems, you'd still want to be able to try to fit all the models, and later look back at those that failed:

```
# If any model fails, all models fail to fit:
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
# If a model fails, it will get a NULL value
models <- lapply(datasets, failwith(NULL, glm),
  formula = y ~ x1 + x2 * x3)

# remove failed models (NULLs) with compact
ok_models <- compact(models)
# extract the datasets corresponding to failed models
failed_data <- datasets[vapply(models, is.null, logical(1))]
```

I think this is a great example of the power of combining functionals and function operators: it lets you succinctly express what you need to solve a common data analysis problem.

12.2.2 Changing what a function does

Other output function operators can have a more profound effect on the operation of the function. Instead of returning the original return value, we can return some other effect of the function evaluation. Here are two examples:

- Return text that the function `print()`ed:

```
capture_it <- function(f) {
  force(f)
  function(...) {
    capture.output(f(...))
  }
}
str_out <- capture_it(str)
str(1:10)
#>  int [1:10] 1 2 3 4 5 6 7 8 9 10
str_out(1:10)
#> [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

- Return how long a function took to run:

```
time_it <- function(f) {
  force(f)
  function(...) {
    system.time(f(...))
  }
}
```

`time_it()` allows us to rewrite some of the code from the functionals chapter:

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x)
)
x <- runif(1e6)

# Previously we used an anonymous function to time execution:
# lapply(compute_mean, function(f) system.time(f(x)))

# Now we can compose function operators:
```

```
call_fun <- function(f, ...) f(...)
lapply(compute_mean, time_it(call_fun), x)
#> $base
#>    user  system elapsed
#>   0.002   0.000   0.002
#>
#> $sum
#>    user  system elapsed
#>   0.002   0.000   0.002
```

In this example, there's not a huge benefit to using function operators, because the composition is simple and we're applying the same operator to each function. Generally, using function operators is most effective when you are using multiple operators or if the gap between creating them and using them is large.

12.2.3 Exercises

1. Create a `negative()` FO that flips the sign of the output of the function to which it is applied.
2. The `evaluate` package makes it easy to capture all the outputs (results, text, messages, warnings, errors, and plots) from an expression. Create a function like `capture_it()` that also captures the warnings and errors generated by a function.
3. Create a FO that tracks files created or deleted in the working directory (Hint: use `dir()` and `setdiff()`.) What other global effects of functions might you want to track?

12.3 Input FOs

The next step up in complexity is to modify the inputs of a function. Again, you can modify how a function works in a minor way (e.g., setting default argument values), or in a major way (e.g., converting inputs from scalars to vectors, or vectors to matrices).

12.3.1 Prefilling function arguments: partial function application

A common use of anonymous functions is to make a variant of a function that has certain arguments "filled in" already. This is called "partial function application", and is implemented by `pryr::partial()`. Once you have read Chapter 14, I encourage you to read the source code for `partial()` and figure out how it works — it's only 5 lines of code!

`partial()` allows us to replace code like

```
f <- function(a) g(a, b = 1)
compact <- function(x) Filter(Negate(is.null), x)
Map(function(x, y) f(x, y, zs), xs, ys)
```

with

```
f <- partial(g, b = 1)
compact <- partial(Filter, Negate(is.null))
Map(partial(f, zs = zs), xs, ys)
```

We can use this idea to simplify the code used when working with lists of functions. Instead of:

```
funs2 <- list(
  sum = function(...) sum(..., na.rm = TRUE),
  mean = function(...) mean(..., na.rm = TRUE),
  median = function(...) median(..., na.rm = TRUE)
)
```

we can write:

```
library(pryr)
funs2 <- list(
  sum = partial(sum, na.rm = TRUE),
  mean = partial(mean, na.rm = TRUE),
  median = partial(median, na.rm = TRUE)
)
```

Using partial function application is a straightforward task in many functional programming languages, but it's not entirely clear how it should

interact with R's lazy evaluation rules. The approach `plyr::partial()` takes is to create a function that is as similar as possible to the anonymous function that you'd create by hand. Peter Meilstrup takes a different approach in his ptools package (`https://github.com/crowding/ptools/`). If you're interested in the topic, you might want to read about the binary operators he created: `%()%`, `%>>%`, and `%<<%`.

12.3.2 Changing input types

It's also possible to make a major change to a function's input, making a function work with fundamentally different types of data. There are a few existing functions that work along these lines:

- `base::Vectorize()` converts a scalar function to a vector function. It takes a non-vectorised function and vectorises it with respect to the arguments specified in the `vectorize.args` argument. This doesn't give you any magical performance improvements, but it's useful if you want a quick and dirty way of making a vectorised function.

  A mildly useful extension to `sample()` would be to vectorize it with respect to size. Doing so would allow you to generate multiple samples in one call.

  ```
  sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
  str(sample2(1:5, c(1, 1, 3)))
  #> List of 3
  #>  $ : int 2
  #>  $ : int 1
  #>  $ : int [1:3] 2 1 5
  str(sample2(1:5, 5:3))
  #> List of 3
  #>  $ : int [1:5] 5 1 2 3 4
  #>  $ : int [1:4] 2 3 5 1
  #>  $ : int [1:3] 5 4 2
  ```

  In this example we have used `SIMPLIFY = FALSE` to ensure that our newly vectorised function always returns a list. This is usually what you want.

- `splat()` converts a function that takes multiple arguments to a function that takes a single list of arguments.

  ```
  splat <- function (f) {
  ```

```
  force(f)
  function(args) {
    do.call(f, args)
  }
}
```

This is useful if you want to invoke a function with varying arguments:

```
x <- c(NA, runif(100), 1000)
args <- list(
  list(x),
  list(x, na.rm = TRUE),
  list(x, na.rm = TRUE, trim = 0.1)
)
lapply(args, splat(mean))
#> [[1]]
#> [1] NA
#>
#> [[2]]
#> [1] 10.4
#>
#> [[3]]
#> [1] 0.478
```

- `plyr::colwise()` converts a vector function to one that works with data frames:

```
median(mtcars)
#> Error in median.default(mtcars): need numeric data
median(mtcars$mpg)
#> [1] 19.2
plyr::colwise(median)(mtcars)
#>    mpg cyl disp  hp drat   wt qsec vs am gear carb
#> 1 19.2   6  196 123  3.7 3.33 17.7  0  0    4    2
```

12.3.3 Exercises

1. Our previous `download()` function only downloads a single file. How can you use `partial()` and `lapply()` to create a function that downloads multiple files at once? What are the pros and cons of using `partial()` vs. writing a function by hand?
2. Read the source code for `plyr::colwise()`. How does the code work? What are `colwise()`'s three main tasks? How could

you make `colwise()` simpler by implementing each task as a function operator? (Hint: think about `partial()`.)

3. Write FOs that convert a function to return a matrix instead of a data frame, or a data frame instead of a matrix. If you understand S3, call them `as.data.frame.function()` and `as.matrix.function()`.
4. You've seen five functions that modify a function to change its output from one form to another. What are they? Draw a table of the various combinations of types of outputs: what should go in the rows and what should go in the columns? What function operators might you want to write to fill in the missing cells? Come up with example use cases.
5. Look at all the examples of using an anonymous function to partially apply a function in this and the previous chapter. Replace the anonymous function with `partial()`. What do you think of the result? Is it easier or harder to read?

12.4 Combining FOs

Besides just operating on single functions, function operators can take multiple functions as input. One simple example of this is `plyr::each()`. It takes a list of vectorised functions and combines them into a single function.

```
summaries <- plyr::each(mean, sd, median)
summaries(1:10)
#>   mean     sd median
#>   5.50   3.03   5.50
```

Two more complicated examples are combining functions through composition, or through boolean algebra. These capabilities are the glue that allow us to join multiple functions together.

12.4.1 Function composition

An important way of combining functions is through composition: `f(g(x))`. Composition takes a list of functions and applies them

sequentially to the input. It's a replacement for the common pattern of anonymous function that chains multiple functions together to get the result you want:

```
sapply(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

A simple version of compose looks like this:

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

(`pryr::compose()` provides a more full-featured alternative that can accept multiple functions and is used for the rest of the examples.)

This allows us to write:

```
sapply(mtcars, compose(length, unique))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Mathematically, function composition is often denoted with the infix operator, o, `(f o g)(x)`. Haskell, a popular functional programming language, uses `.` to the same end. In R, we can create our own infix composition function:

```
"%o%" <- compose
sapply(mtcars, length %o% unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6

sqrt(1 + 8)
#> [1] 3
compose(sqrt, `+`)(1, 8)
#> [1] 3
(sqrt %o% `+`)(1, 8)
#> [1] 3
```

Compose also allows for a very succinct implementation of `Negate`, which is just a partially evaluated version of `compose()`.

```
Negate <- partial(compose, `!`)
```

We could implement the population standard deviation with function composition:

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)

sd2 <- sqrt %o% mean %o% square %o% deviation
sd2(1:10)
#> [1] 2.87
```

This type of programming is called tacit or point-free programming. (The term point-free comes from the use of "point" to refer to values in topology; this style is also derogatorily known as pointless). In this style of programming, you don't explicitly refer to variables. Instead, you focus on the high-level composition of functions rather than the low-level flow of data. The focus is on what's being done, not on objects it's being done to. Since we're using only functions and not parameters, we use verbs and not nouns. This style is common in Haskell, and is the typical style in stack based programming languages like Forth and Factor. It's not a terribly natural or elegant style in R, but it is fun to play with.

`compose()` is particularly useful in conjunction with `partial()`, because `partial()` allows you to supply additional arguments to the functions being composed. One nice side effect of this style of programming is that it keeps a function's arguments near its name. This is important because as the size of the chunk of code you have to hold in your head grows code becomes harder to understand.

Below I take the example from the first section of the chapter and modify it to use the two styles of function composition described above. Both results are longer than the original code, but they may be easier to understand because the function and its arguments are closer together. Note that we still have to read them from right to left (bottom to top): the first function called is the last one written. We could define `compose()` to work in the opposite direction, but in the long run, this is likely to lead to confusion since we'd create a small part of the langugage that reads differently from every other part.

```
download <- dot_every(10, memoise(delay_by(1, download_file)))
```

```
download <- pryr::compose(
  partial(dot_every, 10),
  memoise,
  partial(delay_by, 1),
  download_file
)

download <- partial(dot_every, 10) %o%
  memoise %o%
  partial(delay_by, 1) %o%
  download_file
```

12.4.2 Logical predicates and boolean algebra

When I use `Filter()` and other functionals that work with logical predicates, I often find myself using anonymous functions to combine multiple conditions:

```
Filter(function(x) is.character(x) || is.factor(x), iris)
```

As an alternative, we could define function operators that combine logical predicates:

```
and <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) && f2(...)
  }
}

or <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) || f2(...)
  }
}

not <- function(f) {
  force(f)
  function(...) {
    !f(...)
```

```
  }
}
```

This would allow us to write:

```
Filter(or(is.character, is.factor), iris)
Filter(not(is.numeric), iris)
```

And we now have a boolean algebra on functions, not on the results of functions.

12.4.3 Exercises

1. Implement your own version of `compose()` using `Reduce` and `%o%`. For bonus points, do it without calling `function`.
2. Extend `and()` and `or()` to deal with any number of input functions. Can you do it with `Reduce()`? Can you keep them lazy (e.g., for `and()`, the function returns once it sees the first `FALSE`)?
3. Implement the `xor()` binary operator. Implement it using the existing `xor()` function. Implement it as a combination of `and()` and `or()`. What are the advantages and disadvantages of each approach? Also think about what you'll call the resulting function to avoid a clash with the existing `xor()` function, and how you might change the names of `and()`, `not()`, and `or()` to keep them consistent.
4. Above, we implemented boolean algebra for functions that return a logical function. Implement elementary algebra (`plus()`, `minus()`, `multiply()`, `divide()`, `exponentiate()`, `log()`) for functions that return numeric vectors.

Part III

Computing on the language

13

Non-standard evaluation

> "Flexibility in syntax, if it does not lead to ambiguity, would seem a reasonable thing to ask of an interactive programming language."
>
> — Kent Pitman

R has powerful tools for computing not only on values, but also on the actions that lead to those values. If you're coming from another programming language, they are one of the most surprising features of R. Consider the following simple snippet of code that plots a sine curve:

```
x <- seq(0, 2 * pi, length = 100)
sinx <- sin(x)
plot(x, sinx, type = "l")
```

Look at the labels on the axes. How did R know that the variable on the x axis is called `x` and the variable on the y axis is called `sinx`? In most programming languages, you can only access the values of a function's arguments. In R, you can also access the code used to compute them. This makes it possible to evaluate code in non-standard ways: to use what is known as **non-standard evaluation**, or NSE for short. NSE is particularly useful for functions when doing interactive data analysis because it can dramatically reduce the amount of typing.

Outline

- Section 13.1 teaches you how to capture unevaluated expressions using `substitute()`.
- Section 13.2 shows you `subset()` works with combining `substitute()` with `eval()` to allow you to succinctly select rows from a data frame.
- Section 13.3 discusses scoping issues specific to NSE, and will show you how to resolve them.

- Section 13.4 shows why every function that uses NSE should have an escape hatch, a version that uses regular evaluation.
- Section 13.5 teaches you how to use `substitute()` to work with functions that don't have an escape hatch.
- Section 13.6 finishes off the chapter with a discussion of the downsides of NSE.

Prerequisites

Before reading this chapter, make sure you're familiar with environments (Chapter 8) and lexical scoping (Section 6.2). You'll also need to install the pryr package with `install.packages("pryr")`. Some exercises require the plyr package, which you can install from CRAN with `install.packages("plyr")`.

13.1 Capturing expressions

`substitute()` makes non-standard evaluation possible. It looks at a function argument and instead of seeing the value, it sees the code used to compute the value:

```
f <- function(x) {
  substitute(x)
}
f(1:10)
#> 1:10

x <- 10
f(x)
#> x

y <- 13
f(x + y^2)
#> x + y^2
```

For now, we won't worry about exactly what `substitute()` returns (that's the topic of Chapter 14), but we'll call it an expression.

`substitute()` works because function arguments are represented by a special type of object called a **promise**. A promise captures the expression needed to compute the value and the environment in which to compute it. You're not normally aware of promises because the first time you access a promise its code is evaluated in its environment, yielding a value.

`substitute()` is often paired with `deparse()`. That function takes the result of `substitute()`, an expression, and turns it into a character vector.

```
g <- function(x) deparse(substitute(x))
g(1:10)
#> [1] "1:10"
g(x)
#> [1] "x"
g(x + y^2)
#> [1] "x + y^2"
```

There are a lot of functions in Base R that use these ideas. Some use them to avoid quotes:

```
library(ggplot2)
# the same as
library("ggplot2")
```

Other functions, like `plot.default()`, use them to provide default labels. `data.frame()` labels variables with the expression used to compute them:

```
x <- 1:4
y <- letters[1:4]
names(data.frame(x, y))
#> [1] "x" "y"
```

We'll learn about the ideas that underlie all these examples by looking at one particularly useful application of NSE: `subset()`.

13.1.1 Exercises

1. One important feature of `deparse()` to be aware of when programming is that it can return multiple strings if the input is too long. For example, the following call produces a vector of length two:

```
g(a + b + c + d + e + f + g + h + i + j + k + l + m +
  n + o + p + q + r + s + t + u + v + w + x + y + z)
```

Why does this happen? Carefully read the documentation. Can you write a wrapper around `deparse()` so that it always returns a single string?

2. Why does `as.Date.default()` use `substitute()` and `deparse()`? Why does `pairwise.t.test()` use them? Read the source code.

3. `pairwise.t.test()` assumes that `deparse()` always returns a length one character vector. Can you construct an input that violates this expectation? What happens?

4. `f()`, defined above, just calls `substitute()`. Why can't we use it to define `g()`? In other words, what will the following code return? First make a prediction. Then run the code and think about the results.

```
f <- function(x) substitute(x)
g <- function(x) deparse(f(x))
g(1:10)
g(x)
g(x + y ^ 2 / z + exp(a * sin(b)))
```

13.2 Non-standard evaluation in subset

While printing out the code supplied to an argument value can be useful, we can actually do more with the unevaluated code. Take `subset()`, for example. It's a useful interactive shortcut for subsetting data frames: instead of repeating the name of data frame many times, you can save some typing:

```
sample_df <- data.frame(a = 1:5, b = 5:1, c = c(5, 3, 1, 4, 1))

subset(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1
# equivalent to:
# sample_df[sample_df$a >= 4, ]
```

```
subset(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
# equivalent to:
# sample_df[sample_df$b == sample_df$c, ]
```

`subset()` is special because it implements different scoping rules: the expressions `a >= 4` or `b == c` are evaluated in the specified data frame rather than in the current or global environments. This is the essence of non-standard evaluation.

How does `subset()` work? We've already seen how to capture an argument's expression rather than its result, so we just need to figure out how to evaluate that expression in the right context. Specifically, we want `x` to be interpreted as `sample_df$x`, not `globalenv()$x`. To do this, we need `eval()`. This function takes an expression and evaluates it in the specified environment.

Before we can explore `eval()`, we need one more useful function: `quote()`. It captures an unevaluated expression like `substitute()`, but doesn't do any of the advanced transformations that can make `substitute()` confusing. `quote()` always returns its input as is:

```
quote(1:10)
#> 1:10
quote(x)
#> x
quote(x + y^2)
#> x + y^2
```

We need `quote()` to experiment with `eval()` because `eval()`'s first argument is an expression. So if you only provide one argument, it will evaluate the expression in the current environment. This makes `eval(quote(x))` exactly equivalent to `x`, regardless of what `x` is:

```
eval(quote(x <- 1))
eval(quote(x))
#> [1] 1

eval(quote(y))
#> Error in eval(expr, envir, enclos): object 'y' not found
```

`quote()` and `eval()` are opposites. In the example below, each `eval()` peels off one layer of `quote()`'s.

```
quote(2 + 2)
#> 2 + 2
eval(quote(2 + 2))
#> [1] 4

quote(quote(2 + 2))
#> quote(2 + 2)
eval(quote(quote(2 + 2)))
#> 2 + 2
eval(eval(quote(quote(2 + 2))))
#> [1] 4
```

`eval()`'s second argument specifies the environment in which the code is executed:

```
x <- 10
eval(quote(x))
#> [1] 10

e <- new.env()
e$x <- 20
eval(quote(x), e)
#> [1] 20
```

Because lists and data frames bind names to values in a similar way to environments, `eval()`'s second argument need not be limited to an environment: it can also be a list or a data frame.

```
eval(quote(x), list(x = 30))
#> [1] 30
eval(quote(x), data.frame(x = 40))
#> [1] 40
```

This gives us one part of `subset()`:

```
eval(quote(a >= 4), sample_df)
#> [1] FALSE FALSE FALSE  TRUE  TRUE
eval(quote(b == c), sample_df)
#> [1]  TRUE FALSE FALSE FALSE  TRUE
```

A common mistake when using `eval()` is to forget to quote the first argument. Compare the results below:

```
a <- 10
eval(quote(a), sample_df)
#> [1] 1 2 3 4 5
eval(a, sample_df)
#> [1] 10

eval(quote(b), sample_df)
#> [1] 5 4 3 2 1
eval(b, sample_df)
#> Error in eval(b, sample_df): object 'b' not found
```

We can use `eval()` and `substitute()` to write `subset()`. We first capture the call representing the condition, then we evaluate it in the context of the data frame and, finally, we use the result for subsetting:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x)
  x[r, ]
}
subset2(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1
```

13.2.1 Exercises

1. Predict the results of the following lines of code:

   ```
   eval(quote(eval(quote(eval(quote(2 + 2))))))
   eval(eval(quote(eval(quote(eval(quote(2 + 2)))))))
   quote(eval(quote(eval(quote(eval(quote(2 + 2)))))))
   ```

2. `subset2()` has a bug if you use it with a single column data frame. What should the following code return? How can you modify `subset2()` so it returns the correct type of object?

   ```
   sample_df2 <- data.frame(x = 1:10)
   subset2(sample_df2, x > 8)
   #> [1]  9 10
   ```

3. The real subset function (`subset.data.frame()`) removes missing values in the condition. Modify `subset2()` to do the same: drop the offending rows.
4. What happens if you use `quote()` instead of `substitute()` inside of `subset2()`?
5. The second argument in `subset()` allows you to select variables. It treats variable names as if they were positions. This allows you to do things like `subset(mtcars, , -cyl)` to drop the cylinder variable, or `subset(mtcars, , disp:drat)` to select all the variables between `disp` and `drat`. How does this work? I've made this easier to understand by extracting it out into its own function.

   ```
   select <- function(df, vars) {
     vars <- substitute(vars)
     var_pos <- setNames(as.list(seq_along(df)), names(df))
     pos <- eval(vars, var_pos)
     df[, pos, drop = FALSE]
   }
   select(mtcars, -cyl)
   ```

6. What does `evalq()` do? Use it to reduce the amount of typing for the examples above that use both `eval()` and `quote()`.

13.3 Scoping issues

It certainly looks like our `subset2()` function works. But since we're working with expressions instead of values, we need to test things more extensively. For example, the following applications of `subset2()` should all return the same value because the only difference between them is the name of a variable:

```
y <- 4
x <- 4
condition <- 4
condition_call <- 4

subset2(sample_df, a == 4)
#>   a b c
```

```
#> 4 4 2 4
subset2(sample_df, a == y)
#>   a b c
#> 4 4 2 4
subset2(sample_df, a == x)
#>       a  b  c
#> 1     1  5  5
#> 2     2  4  3
#> 3     3  3  1
#> 4     4  2  4
#> 5     5  1  1
#> NA   NA NA NA
#> NA.1 NA NA NA
subset2(sample_df, a == condition)
#> Error in eval(expr, envir, enclos): object 'a' not found
subset2(sample_df, a == condition_call)
#> Warning in a == condition_call: longer object length is not a
#> multiple of shorter object length
#> [1] a b c
#> <0 rows> (or 0-length row.names)
```

What went wrong? You can get a hint from the variable names I've chosen: they are all names of variables defined inside `subset2()`. If `eval()` can't find the variable inside the data frame (its second argument), it looks in the environment of `subset2()`. That's obviously not what we want, so we need some way to tell `eval()` where to look if it can't find the variables in the data frame.

The key is the third argument to `eval()`: `enclos`. This allows us to specify a parent (or enclosing) environment for objects that don't have one (like lists and data frames). If the binding is not found in `env`, `eval()` will next look in `enclos`, and then in the parents of `enclos`. `enclos` is ignored if `env` is a real environment. We want to look for `x` in the environment from which `subset2()` was called. In R terminology this is called the **parent frame** and is accessed with `parent.frame()`. This is an example of dynamic scope (`http://en.wikipedia.org/wiki/Scope_%28programming%29#Dynamic_scoping`): the values come from the location where the function was called, not where it was defined.

With this modification our function now works:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
```

```
  x[r, ]
}

x <- 4
subset2(sample_df, a == x)
#>   a b c
#> 4 4 2 4
```

Using `enclos` is just a shortcut for converting a list or data frame to an environment. We can get the same behaviour by using `list2env()`. It turns a list into an environment with an explicit parent:

```
subset2a <- function(x, condition) {
  condition_call <- substitute(condition)
  env <- list2env(x, parent = parent.frame())
  r <- eval(condition_call, env)
  x[r, ]
}

x <- 5
subset2a(sample_df, a == x)
#>   a b c
#> 5 5 1 1
```

13.3.1 Exercises

1. `plyr::arrange()` works similarly to `subset()`, but instead of selecting rows, it reorders them. How does it work? What does `substitute(order(...))` do? Create a function that does only that and experiment with it.
2. What does `transform()` do? Read the documentation. How does it work? Read the source code for `transform.data.frame()`. What does `substitute(list(...))` do?
3. `plyr::mutate()` is similar to `transform()` but it applies the transformations sequentially so that transformation can refer to columns that were just created:

   ```
   df <- data.frame(x = 1:5)
   transform(df, x2 = x * x, x3 = x2 * x)
   plyr::mutate(df, x2 = x * x, x3 = x2 * x)
   ```

How does mutate work? What's the key difference between `mutate()` and `transform()`?

4. What does `with()` do? How does it work? Read the source code for `with.default()`. What does `within()` do? How does it work? Read the source code for `within.data.frame()`. Why is the code so much more complex than `with()`?

13.4 Calling from another function

Typically, computing on the language is most useful when functions are called directly by users and less useful when they are called by other functions. While `subset()` saves typing, it's actually difficult to use non-interactively. For example, imagine we want to create a function that randomly reorders a subset of rows of data. A nice way to do that would be to compose a function that reorders with another that selects. Let's try that:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

scramble <- function(x) x[sample(nrow(x)), ]

subscramble <- function(x, condition) {
  scramble(subset2(x, condition))
}
```

But it doesn't work:

```
subscramble(sample_df, a >= 4)
# Error in eval(expr, envir, enclos) : object 'a' not found
traceback()
#> 5: eval(expr, envir, enclos)
#> 4: eval(condition_call, x, parent.frame()) at #3
#> 3: subset2(x, condition) at #1
#> 2: scramble(subset2(x, condition)) at #2
#> 1: subscramble(sample_df, a >= 4)
```

What's gone wrong? To figure it out, let us `debug()` subset2()‘ and work through the code line-by-line:

```
debugonce(subset2)
subscramble(sample_df, a >= 4)
#> debugging in: subset2(x, condition)
#> debug at #1: {
#>     condition_call <- substitute(condition)
#>     r <- eval(condition_call, x, parent.frame())
#>     x[r, ]
#> }
n
#> debug at #2: condition_call <- substitute(condition)
n
#> debug at #3: r <- eval(condition_call, x, parent.frame())
r <- eval(condition_call, x, parent.frame())
#> Error in eval(expr, envir, enclos) : object 'a' not found
condition_call
#> condition
eval(condition_call, x)
#> Error in eval(expr, envir, enclos) : object 'a' not found
Q
```

Can you see what the problem is? `condition_call` contains the expression `condition`. So when we evaluate `condition_call` it also evaluates `condition`, which has the value `a >= 4`. However, this can't be computed because there's no object called `a` in the parent environment. But, if `a` were set in the global environment, even more confusing things can happen:

```
a <- 4
subscramble(sample_df, a == 4)
#>   a b c
#> 1 1 5 5
#> 4 4 2 4
#> 2 2 4 3
#> 5 5 1 1
#> 3 3 3 1

a <- c(1, 1, 4, 4, 4, 4)
subscramble(sample_df, a >= 4)
#>     a  b  c
```

```
#> 4   4  2  4
#> NA NA NA NA
#> 3   3  3  1
#> 5   5  1  1
```

This is an example of the general tension between functions that are designed for interactive use and functions that are safe to program with. A function that uses `substitute()` might reduce typing, but it can be difficult to call from another function.

As a developer, you should always provide an escape hatch: an alternative version of the function that uses standard evaluation. In this case, we could write a version of `subset2()` that takes an already quoted expression:

```
subset2_q <- function(x, condition) {
  r <- eval(condition, x, parent.frame())
  x[r, ]
}
```

Here I use the suffix `_q` to indicate that it takes a quoted expression. Most users won't need this function so the name can be a little longer.

We can then rewrite both `subset2()` and `subscramble()` to use `subset2_q()`:

```
subset2 <- function(x, condition) {
  subset2_q(x, substitute(condition))
}

subscramble <- function(x, condition) {
  condition <- substitute(condition)
  scramble(subset2_q(x, condition))
}

subscramble(sample_df, a >= 3)
#>   a b c
#> 5 5 1 1
#> 4 4 2 4
#> 3 3 3 1
subscramble(sample_df, a >= 3)
#>   a b c
#> 3 3 3 1
```

```
#> 5 5 1 1
#> 4 4 2 4
```

Base R functions tend to use a different sort of escape hatch. They often have an argument that turns off NSE. For example, `require()` has `character.only = TRUE`. I don't think it's a good idea to use an argument to change the behaviour of another argument because it makes function calls harder to understand.

13.4.1 Exercises

1. The following R functions all use NSE. For each, describe how it uses NSE, and read the documentation to determine its escape hatch.

   - `rm()`
   - `library()` and `require()`
   - `substitute()`
   - `data()`
   - `data.frame()`

2. Base functions `match.fun()`, `page()`, and `ls()` all try to automatically determine whether you want standard or non-standard evaluation. Each uses a different approach. Figure out the essence of each approach then compare and contrast.
3. Add an escape hatch to `plyr::mutate()` by splitting it into two functions. One function should capture the unevaluated inputs. The other should take a data frame and list of expressions and perform the computation.
4. What's the escape hatch for `ggplot2::aes()`? What about `plyr::()`? What do they have in common? What are the advantages and disadvantages of their differences?
5. The version of `subset2_q()` I presented is a simplification of real code. Why is the following version better?

   ```
   subset2_q <- function(x, cond, env = parent.frame()) {
     r <- eval(cond, x, env)
     x[r, ]
   }
   ```

   Rewrite `subset2()` and `subscramble()` to use this improved version.

13.5 Substitute

Most functions that use non-standard evaluation provide an escape hatch. But what happens if you want to call a function that doesn't have one? For example, imagine you want to create a lattice graphic given the names of two variables:

```
library(lattice)
xyplot(mpg ~ disp, data = mtcars)

x <- quote(mpg)
y <- quote(disp)
xyplot(x ~ y, data = mtcars)
#> Error in tmp[subset]: object of type 'symbol' is not subsettable
```

We might turn to `substitute()` and use it for another purpose: to modify an expression. Unfortunately `substitute()` has a feature that makes modifying calls interactively a bit of a pain. When run from the global environment, it never does substitutions: in fact, in this situation it behaves just like `quote()`:

```
a <- 1
b <- 2
substitute(a + b + z)
#> a + b + z
```

However, if you run it inside a function, `substitute()` does substitute and leaves everything else as is:

```
f <- function() {
  a <- 1
  b <- 2
  substitute(a + b + z)
}
f()
#> 1 + 2 + z
```

To make it easier to experiment with `substitute()`, `pryr` provides the `subs()` function. It works exactly the same way as `substitute()` except it has a shorter name and it works in the global environment. These two features make experimentation easier:

```
a <- 1
b <- 2
subs(a + b + z)
#> 1 + 2 + z
```

The second argument (of both `subs()` and `substitute()`) can override the use of the current environment, and provide an alternative via a list of name-value pairs. The following example uses this technique to show some variations on substituting a string, variable name, or function call:

```
subs(a + b, list(a = "y"))
#> "y" + b
subs(a + b, list(a = quote(y)))
#> y + b
subs(a + b, list(a = quote(y())))
#> y() + b
```

Remember that every action in R is a function call, so we can also replace `+` with another function:

```
subs(a + b, list("+" = quote(f)))
#> f(a, b)
subs(a + b, list("+" = quote(`*`)))
#> a * b
```

You can also make nonsense code:

```
subs(y <- y + 1, list(y = 1))
#> 1 <- 1 + 1
```

Formally, substitution takes place by examining all the names in the expression. If the name refers to:

1. an ordinary variable, it's replaced by the value of the variable.
2. a promise (a function argument), it's replaced by the expression associated with the promise.
3. ..., it's replaced by the contents of ....

Otherwise it's left as is.

We can use this to create the right call to `xyplot()`:

```
x <- quote(mpg)
y <- quote(disp)
subs(xyplot(x ~ y, data = mtcars))
#> xyplot(mpg ~ disp, data = mtcars)
```

It's even simpler inside a function, because we don't need to explicitly quote the x and y variables (rule 2 above):

```
xyplot2 <- function(x, y, data = data) {
  substitute(xyplot(x ~ y, data = data))
}
xyplot2(mpg, disp, data = mtcars)
#> xyplot(mpg ~ disp, data = mtcars)
```

If we include `...` in the call to substitute, we can add additional arguments to the call:

```
xyplot3 <- function(x, y, ...) {
  substitute(xyplot(x ~ y, ...))
}
xyplot3(mpg, disp, data = mtcars, col = "red", aspect = "xy")
#> xyplot(mpg ~ disp, data = mtcars, col = "red", aspect = "xy")
```

To create the plot, we'd then `eval()` this call.

13.5.1 Adding an escape hatch to substitute

`substitute()` is itself a function that uses non-standard evaluation and doesn't have an escape hatch. This means we can't use `substitute()` if we already have an expression saved in a variable:

```
x <- quote(a + b)
substitute(x, list(a = 1, b = 2))
#> x
```

Although `substitute()` doesn't have a built-in escape hatch, we can use the function itself to create one:

```
substitute_q <- function(x, env) {
  call <- substitute(substitute(y, env), list(y = x))
```

```
  eval(call)
}

x <- quote(a + b)
substitute_q(x, list(a = 1, b = 2))
#> 1 + 2
```

The implementation of `substitute_q()` is short, but deep. Let's work through the example above: `substitute_q(x, list(a = 1, b = 2))`. It's a little tricky because `substitute()` uses NSE so we can't use the usual technique of working through the parentheses inside-out.

1. First `substitute(substitute(y, env), list(y = x))` is evaluated. The expression `substitute(y, env)` is captured and `y` is replaced by the value of `x`. Because we've put `x` inside a list, it will be evaluated and the rules of substitute will replace `y` with its value. This yields the expression `substitute(a + b, env)`
2. Next we evaluate that expression inside the current function. `substitute()` evaluates its first argument, and looks for name value pairs in `env`. Here, it evaluates as `list(a = 1, b = 2)`. Since these are both values (not promises), the result will be `1 + 2`

A slightly more rigorous version of `substitute_q()` is provided by the pryr package.

13.5.2 Capturing unevaluated ...

Another useful technique is to capture all of the unevaluated expressions in `...`. Base R functions do this in many ways, but there's one technique that works well across a wide variety of situations:

```
dots <- function(...) {
  eval(substitute(alist(...)))
}
```

This uses the `alist()` function which simply captures all its arguments. This function is the same as `pryr::dots()`. Pryr also provides `pryr::named_dots()`, which, by using deparsed expressions as default names, ensures that all arguments are named (just like `data.frame()`).

13.5.3 Exercises

1. Use `subs()` to convert the LHS to the RHS for each of the following pairs:
   - `a + b + c -> a * b * c`
   - `f(g(a, b), c) -> (a + b) * c`
   - `f(a < b, c, d) -> if (a < b) c else d`
2. For each of the following pairs of expressions, describe why you can't use `subs()` to convert one to the other.
   - `a + b + c -> a + b * c`
   - `f(a, b) -> f(a, b, c)`
   - `f(a, b, c) -> f(a, b)`
3. How does `pryr::named_dots()` work? Read the source.

13.6 The downsides of non-standard evaluation

The biggest downside of NSE is that functions that use it are no longer referentially transparent (`http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)`). A function is **referentially transparent** if you can replace its arguments with their values and its behaviour doesn't change. For example, if a function, `f()`, is referentially transparent and both `x` and `y` are 10, then `f(x)`, `f(y)`, and `f(10)` will all return the same result. Referentially transparent code is easier to reason about because the names of objects don't matter, and because you can always work from the innermost parentheses outwards.

There are many important functions that by their very nature are not referentially transparent. Take the assignment operator. You can't take `a <- 1` and replace `a` by its value and get the same behaviour. This is one reason that people usually write assignments at the top-level of functions. It's hard to reason about code like this:

```
a <- 1
b <- 2
if ((b <- a + 1) > (a <- b - 1)) {
  b <- b + 2
}
```

Using NSE prevents a function from being referentially transparent. This makes the mental model needed to correctly predict the output much more complicated. So, it's only worthwhile to use NSE if there is significant gain. For example, `library()` and `require()` can be called either with or without quotes, because internally they use `deparse(substitute(x))` plus some other tricks. This means that these two lines do exactly the same thing:

```
library(ggplot2)
library("ggplot2")
```

Things start to get complicated if the variable is associated with a value. What package will this load?

```
ggplot2 <- "plyr"
library(ggplot2)
```

There are a number of other R functions that work in this way, like `ls()`, `rm()`, `data()`, `demo()`, `example()`, and `vignette()`. To me, eliminating two keystrokes is not worth the loss of referential transparency, and I don't recommend you use NSE for this purpose.

One situation where non-standard evaluation is worthwhile is `data.frame()`. If not explicitly supplied, it uses the input to automatically name the output variables:

```
x <- 10
y <- "a"
df <- data.frame(x, y)
names(df)
#> [1] "x" "y"
```

I think it's worthwhile because it eliminates a lot of redundancy in the common scenario when you're creating a data frame from existing variables. More importantly, if needed, it's easy to override this behaviour by supplying names for each variable.

Non-standard evaluation allows you to write functions that are extremely powerful. However, they are harder to understand and to program with. As well as always providing an escape hatch, carefully consider both the costs and benefits of NSE before using it in a new domain.

13.6.1 Exercises

1. What does the following function do? What's the escape hatch? Do you think that this is an appropriate use of NSE?

   ```
   nl <- function(...) {
     dots <- named_dots(...)
     lapply(dots, eval, parent.frame())
   }
   ```

2. Instead of relying on promises, you can use formulas created with `~` to explicitly capture an expression and its environment. What are the advantages and disadvantages of making quoting explicit? How does it impact referential transparency?

3. Read the standard non-standard evaluation rules found at `http://developer.r-project.org/nonstandard-eval.pdf`.

14

Expressions

In Chapter 13, you learned the basics of accessing and evaluating the expressions underlying computation in R. In this chapter, you'll learn how to manipulate these expressions with code. You're going to learn how to metaprogram: how to create programs with other programs!

Outline

- Section 14.1 begins with a deep dive into the structure of expressions. You'll learn about the four components of an expression: constants, names, calls, and pairlists.
- Section 14.2 goes into further details about names.
- Section 14.3 gives more details about calls.
- Section 14.4 takes a minor detour to discuss some common uses of calls in base R.
- Section 14.5 completes the discussion of the four major components of an expression, and shows how you can create functions from their component pieces.
- Section 14.6 discusses how to convert back and forth between expressions and text.
- Section 14.7 concludes the chapter, combining everything you've learned about writing functions that can compute on and modify arbitrary R code.

Prerequisites

Throughout this chapter we're going to use tools from the `pryr` package to help see what's going on. If you don't already have it, install it by running `install.packages("pryr")`.

14.1 Structure of expressions

To compute on the language, we first need to understand the structure of the language. That will require some new vocabulary, some new tools, and some new ways of thinking about R code. The first thing you'll need to understand is the distinction between an operation and a result:

```
x <- 4
y <- x * 10
y
#> [1] 40
```

We want to distinguish the action of multiplying `x` by 10 and assigning that result to `y` from the actual result (40). As we've seen in the previous chapter, we can capture the action with `quote()`:

```
z <- quote(y <- x * 10)
z
#> y <- x * 10
```

`quote()` returns an **expression**: an object that represents an action that can be performed by R. (Unfortunately `expression()` does not return an expression in this sense. Instead, it returns something more like a list of expressions. See Section 14.6 for more details.)

An expression is also called an abstract syntax tree (AST) because it represents the hierarchical tree structure of the code. We'll use `pryr::ast()` to see this more clearly:

```
ast(y <- x * 10)
#> \- ()
#>   \- `<-
#>   \- `y
#>   \- ()
#>     \- `*
#>     \- `x
#>     \-  10
```

There are four possible components of an expression: constants, names, calls, and pairlists.

- **constants** are length one atomic vectors, like `"a"` or `10`. `ast()` displays them as is.

  ```
  ast("a")
  #> \-  "a"
  ast(1)
  #> \-  1
  ast(1L)
  #> \-  1L
  ast(TRUE)
  #> \-  TRUE
  ```

  Quoting a constant returns it unchanged:

  ```
  identical(1, quote(1))
  #> [1] TRUE
  identical("test", quote("test"))
  #> [1] TRUE
  ```

- **names**, or symbols, represent the name of an object rather than its value. `ast()` prefixes names with a backtick.

  ```
  ast(x)
  #> \- `x
  ast(mean)
  #> \- `mean
  ast(`an unusual name`)
  #> \- `an unusual name
  ```

- **calls** represent the action of calling a function. Like lists, calls are recursive: they can contain constants, names, pairlists, and other calls. `ast()` prints `()` and then lists the children. The first child is the function that is called, and the remaining children are the function's arguments.

  ```
  ast(f())
  #> \- ()
  #>   \- `f
  ast(f(1, 2))
  #> \- ()
  #>   \- `f
  #>   \-  1
  #>   \-  2
  ast(f(a, b))
  #> \- ()
  ```

```
#>   \- `f
#>   \- `a
#>   \- `b
ast(f(g(), h(1, a)))
#> \- ()
#>   \- `f
#>   \- ()
#>     \- `g
#>   \- ()
#>     \- `h
#>     \-  1
#>     \- `a
```

As mentioned in Section 6.3, even things that don't look like function calls still have this hierarchical structure:

```
ast(a + b)
#> \- ()
#>   \- `+
#>   \- `a
#>   \- `b
ast(if (x > 1) x else 1/x)
#> \- ()
#>   \- `if
#>   \- ()
#>     \- `>
#>     \- `x
#>     \-  1
#>   \- `x
#>   \- ()
#>     \- `/
#>     \-  1
#>     \- `x
```

- **pairlists**, short for dotted pair lists, are a legacy of R's past. They are only used in one place: the formal arguments of a function. `ast()` prints `[]` at the top-level of a pairlist. Like calls, pairlists are also recursive and can contain constants, names, and calls.

```
ast(function(x = 1, y) x)
#> \- ()
#>   \- `function
#>   \- []
#>     \ x = 1
```

```
#>     \ y =`MISSING
#>   \- `x
#>   \- <srcref>
ast(function(x = 1, y = x * 2) {x / y})
#> \- ()
#>   \- `function
#>   \- []
#>     \ x = 1
#>     \ y =()
#>       \- `*
#>       \- `x
#>       \-  2
#>   \- ()
#>     \- `{
#>     \- ()
#>       \- `/
#>       \- `x
#>       \- `y
#>   \- <srcref>
```

Note that `str()` does not follow these naming conventions when describing objects. Instead, it describes names as symbols and calls as language objects:

```
str(quote(a))
#>  symbol a
str(quote(a + b))
#>  language a + b
```

Using low-level functions, it is possible to create call trees that contain objects other than constants, names, calls, and pairlists. The following example uses `substitute()` to insert a data frame into a call tree. This is a bad idea, however, because the object does not print correctly: the printed call looks like it should return "list" but when evaluated, it returns "data.frame".

```
class_df <- substitute(class(df), list(df = data.frame(x = 10)))
class_df
#> class(list(x = 10))
eval(class_df)
#> [1] "data.frame"
```

Together these four components define the structure of all R code. They are explained in more detail in the following sections.

14.1.1 Exercises

1. There's no existing base function that checks if an element is a valid component of an expression (i.e., it's a constant, name, call, or pairlist). Implement one by guessing the names of the "is" functions for calls, names, and pairlists.
2. `pryr::ast()` uses non-standard evaluation. What's its escape hatch to standard evaluation?
3. What does the call tree of an if statement with multiple else conditions look like?
4. Compare `ast(x + y %+% z)` to `ast(x ^ y %+% z)`. What do they tell you about the precedence of custom infix functions?
5. Why can't an expression contain an atomic vector of length greater than one? Which one of the six types of atomic vector can't appear in an expression? Why?

14.2 Names

Typically, we use `quote()` to capture names. You can also convert a string to a name with `as.name()`. However, this is most useful only when your function receives strings as input. Otherwise it involves more typing than using `quote()`. (You can use `is.name()` to test if an object is a name.)

```
as.name("name")
#> name
identical(quote(name), as.name("name"))
#> [1] TRUE

is.name("name")
#> [1] FALSE
is.name(quote(name))
#> [1] TRUE
is.name(quote(f(name)))
#> [1] FALSE
```

(Names are also called symbols. `as.symbol()` and `is.symbol()` are identical to `as.name()` and `is.name()`.)

Names that would otherwise be invalid are automatically surrounded by backticks:

```
as.name("a b")
#> `a b`
as.name("if")
#> `if`
```

There's one special name that needs a little extra discussion: the empty name. It is used to represent missing arguments. This object behaves strangely. You can't bind it to a variable. If you do, it triggers an error about missing arguments. It's only useful if you want to programmatically create a function with missing arguments.

```
f <- function(x) 10
formals(f)$x
is.name(formals(f)$x)
#> [1] TRUE
as.character(formals(f)$x)
#> [1] ""

missing_arg <- formals(f)$x
# Doesn't work!
is.name(missing_arg)
#> Error in eval(expr, envir, enclos): argument "missing_arg" is missing, with no default
```

To explicitly create it when needed, call `quote()` with a named argument:

```
quote(expr =)
```

14.2.1 Exercises

1. You can use `formals()` to both get and set the arguments of a function. Use `formals()` to modify the following function so that the default value of `x` is missing and `y` is 10.

   ```
   g <- function(x = 20, y) {
     x + y
   }
   ```

2. Write an equivalent to `get()` using `as.name()` and `eval()`. Write an equivalent to `assign()` using `as.name()`, `substitute()`, and `eval()`. (Don't worry about the multiple ways of choosing an environment; assume that the user supplies it explicitly.)

14.3 Calls

A call is very similar to a list. It has `length`, `[[` and `[` methods, and is recursive because calls can contain other calls. The first element of the call is the function that gets called. It's usually the *name* of a function:

```
x <- quote(read.csv("important.csv", row.names = FALSE))
x[[1]]
#> read.csv
is.name(x[[1]])
#> [1] TRUE
```

But it can also be another call:

```
y <- quote(add(10)(20))
y[[1]]
#> add(10)
is.call(y[[1]])
#> [1] TRUE
```

The remaining elements are the arguments. They can be extracted by name or by position.

```
x <- quote(read.csv("important.csv", row.names = FALSE))
x[[2]]
#> [1] "important.csv"
x$row.names
#> [1] FALSE
names(x)
#> [1] ""          ""          "row.names"
```

The length of a call minus 1 gives the number of arguments:

```
length(x) - 1
#> [1] 2
```

14.3.1 Modifying a call

You can add, modify, and delete elements of the call with the standard replacement operators, `$<-` and `[[<-`:

```
y <- quote(read.csv("important.csv", row.names = FALSE))
y$row.names <- TRUE
y$col.names <- FALSE
y
#> read.csv("important.csv", row.names = TRUE, col.names = FALSE)

y[[2]] <- quote(paste0(filename, ".csv"))
y[[4]] <- NULL
y
#> read.csv(paste0(filename, ".csv"), row.names = TRUE)

y$sep <- ","
y
#> read.csv(paste0(filename, ".csv"), row.names = TRUE, sep = ",")
```

Calls also support the `[` method. But use it with care. Removing the first element is unlikely to create a useful call.

```
x[-3] # remove the second argument
#> read.csv("important.csv")
x[-1] # remove the function name - but it's still a call!
#> "important.csv"(row.names = FALSE)
x
#> read.csv("important.csv", row.names = FALSE)
```

If you want a list of the unevaluated arguments (expressions), use explicit coercion:

```
# A list of the unevaluated arguments
as.list(x[-1])
```

```
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

Generally speaking, because R's function calling semantics are so flexible, getting or setting arguments by position is dangerous. For example, even though the values at each position are different, the following three calls all have the same effect:

```
m1 <- quote(read.delim("data.txt", sep = "|"))
m2 <- quote(read.delim(s = "|", "data.txt"))
m3 <- quote(read.delim(file = "data.txt", , "|"))
```

To work around this problem, pryr provides `standardise_call()`. It uses the base `match.call()` function to convert all positional arguments to named arguments:

```
standardise_call(m1)
#> read.delim(file = "data.txt", sep = "|")
standardise_call(m2)
#> read.delim(file = "data.txt", sep = "|")
standardise_call(m3)
#> read.delim(file = "data.txt", sep = "|")
```

14.3.2 Creating a call from its components

To create a new call from its components, you can use `call()` or `as.call()`. The first argument to `call()` is a string which gives a function name. The other arguments are expressions that represent the arguments of the call.

```
call(":", 1, 10)
#> 1:10
call("mean", quote(1:10), na.rm = TRUE)
#> mean(1:10, na.rm = TRUE)
```

`as.call()` is a minor variant of `call()` that takes a single list as input. The first element is a name or call. The subsequent elements are the arguments.

```
as.call(list(quote(mean), quote(1:10)))
#> mean(1:10)
as.call(list(quote(adder(10)), 20))
#> adder(10)(20)
```

14.3.3 Exercises

1. The following two calls look the same, but are actually different:

   ```
   (a <- call("mean", 1:10))
   #> mean(1:10)
   (b <- call("mean", quote(1:10)))
   #> mean(1:10)
   identical(a, b)
   #> [1] FALSE
   ```

   What's the difference? Which one should you prefer?

2. Implement a pure R version of `do.call()`.

3. Concatenating a call and an expression with `c()` creates a list. Implement `concat()` so that the following code works to combine a call and an additional argument.

   ```
   concat(quote(f), a = 1, b = quote(mean(a)))
   #> f(a = 1, b = mean(a))
   ```

4. Since `list()`s don't belong in expressions, we could create a more convenient call constructor that automatically combines lists into the arguments. Implement `make_call()` so that the following code works.

   ```
   make_call(quote(mean), list(quote(x), na.rm = TRUE))
   #> mean(x, na.rm = TRUE)
   make_call(quote(mean), quote(x), na.rm = TRUE)
   #> mean(x, na.rm = TRUE)
   ```

5. How does `mode<-` work? How does it use `call()`?

6. Read the source for `pryr::standardise_call()`. How does it work? Why is `is.primitive()` needed?

7. `standardise_call()` doesn't work so well for the following calls. Why?

```
standardise_call(quote(mean(1:10, na.rm = TRUE)))
#> mean(x = 1:10, na.rm = TRUE)
standardise_call(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
standardise_call(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

8. Read the documentation for `pryr::modify_call()`. How do you think it works? Read the source code.
9. Use `ast()` and experimentation to figure out the three arguments in an `if()` call. Which components are required? What are the arguments to the `for()` and `while()` calls?

14.4 Capturing the current call

Many base R functions use the current call: the expression that caused the current function to be run. There are two ways to capture a current call:

- `sys.call()` captures exactly what the user typed.
- `match.call()` makes a call that only uses named arguments. It's like automatically calling `pryr::standardise_call()` on the result of `sys.call()`

The following example illustrates the difference between the two:

```
f <- function(abc = 1, def = 2, ghi = 3) {
  list(sys = sys.call(), match = match.call())
}
f(d = 2, 2)
#> $sys
#> f(d = 2, 2)
#>
#> $match
#> f(abc = 2, def = 2)
```

Modelling functions often use `match.call()` to capture the call used to create the model. This makes it possible to `update()` a model, re-fitting

the model after modifying some of original arguments. Here's an example of `update()` in action:

```
mod <- lm(mpg ~ wt, data = mtcars)
update(mod, formula = . ~ . + cyl)
#>
#> Call:
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           wt          cyl
#>       39.69        -3.19        -1.51
```

How does `update()` work? We can rewrite it using some tools from pryr to focus on the essence of the algorithm.

```
update_call <- function (object, formula., ...) {
  call <- object$call

  # Use update.formula to deal with formulas like . ~ .
  if (!missing(formula.)) {
    call$formula <- update.formula(formula(object), formula.)
  }

  modify_call(call, dots(...))
}
update_model <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, parent.frame())
}
update_model(mod, formula = . ~ . + cyl)
#>
#> Call:
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           wt          cyl
#>       39.69        -3.19        -1.51
```

The original `update()` has an `evaluate` argument that controls whether the function returns the call or the result. But I think it's better, on principle, that a function returns only one type of object, rather than different types depending on the function's arguments.

This rewrite also allows us to fix a small bug in `update()`: it re-evaluates the call in the global environment, when what we really want is to re-evaluate it in the environment where the model was originally fit — in the formula.

```
f <- function() {
  n <- 3
  lm(mpg ~ poly(wt, n), data = mtcars)
}
mod <- f()
update(mod, data = mtcars)
#> Error in poly(wt, n): object 'n' not found

update_model <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, environment(formula(object)))
}
update_model(mod, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ poly(wt, n), data = mtcars)
#>
#> Coefficients:
#>  (Intercept)  poly(wt, n)1  poly(wt, n)2  poly(wt, n)3
#>       20.091       -29.116         8.636         0.275
```

This is an important principle to remember: if you want to re-run code captured with `match.call()`, you also need to capture the environment in which it was evaluated, usually the `parent.frame()`. The downside to this is that capturing the environment also means capturing any large objects which happen to be in that environment, which prevents their memory from being released. This topic is explored in more detail in **??**.

Some base R functions use `match.call()` where it's not necessary. For example, `write.csv()` captures the call to `write.csv()` and mangles it to call `write.table()` instead:

```
write.csv <- function(...) {
  Call <- match.call(expand.dots = TRUE)
  for (arg in c("append", "col.names", "sep", "dec", "qmethod")) {
    if (!is.null(Call[[arg]])) {
      warning(gettextf("attempt to set '%s' ignored", arg))
```

```
    }
  }
  rn <- eval.parent(Call$row.names)
  Call$append <- NULL
  Call$col.names <- if (is.logical(rn) && !rn) TRUE else NA
  Call$sep <- ","
  Call$dec <- "."
  Call$qmethod <- "double"
  Call[[1L]] <- as.name("write.table")
  eval.parent(Call)
}
```

To fix this, we could implement `write.csv()` using regular function call semantics:

```
write.csv <- function(x, file = "", sep = ",", qmethod = "double",
                      ...) {
  write.table(x = x, file = file, sep = sep, qmethod = qmethod,
    ...)
}
```

This is much easier to understand: it's just calling `write.table()` with different defaults. This also fixes a subtle bug in the original `write.csv()`: `write.csv(mtcars, row = FALSE)` raises an error, but `write.csv(mtcars, row.names = FALSE)` does not. The lesson here is that it's always better to solve a problem with the simplest tool possible.

14.4.1 Exercises

1. Compare and contrast `update_model()` with `update.default()`.
2. Why doesn't `write.csv(mtcars, "mtcars.csv", row = FALSE)` work? What property of argument matching has the original author forgotten?
3. Rewrite `update.formula()` to use R code instead of C code.
4. Sometimes it's necessary to uncover the function that called the function that called the current function (i.e., the grandparent, not the parent). How can you use `sys.call()` or `match.call()` to find this function?

14.5 Pairlists

Pairlists are a holdover from R's past. They behave identically to lists, but have a different internal representation (as a linked list rather than a vector). Pairlists have been replaced by lists everywhere except in function arguments.

The only place you need to care about the difference between a list and a pairlist is if you're going to construct functions by hand. For example, the following function allows you to construct a function from its component pieces: a list of formal arguments, a body, and an environment. The function uses `as.pairlist()` to ensure that the `function()` has the pairlist of `args` it needs.

```
make_function <- function(args, body, env = parent.frame()) {
  args <- as.pairlist(args)

  eval(call("function", args, body), env)
}
```

This function is also available in pryr, where it does a little extra checking of arguments. `make_function()` is best used in conjunction with `alist()`, the **a**rgument list function. `alist()` doesn't evaluate its arguments so that `alist(x = a)` is shorthand for `list(x = quote(a))`.

```
add <- make_function(alist(a = 1, b = 2), quote(a + b))
add(1)
#> [1] 3
add(1, 2)
#> [1] 3

# To have an argument with no default, you need an explicit =
make_function(alist(a = , b = a), quote(a + b))
#> function (a, b = a)
#> a + b
# To take `...` as an argument put it on the LHS of =
make_function(alist(a = , b = , ... =), quote(a + b))
#> function (a, b, ...)
#> a + b
```

`make_function()` has one advantage over using closures to construct functions: with it, you can easily read the source code. For example:

```
adder <- function(x) {
  make_function(alist(y =), substitute({x + y}), parent.frame())
}
adder(10)
#> function (y)
#> {
#>     10 + y
#> }
```

One useful application of `make_function()` is in functions like `curve()`. `curve()` allows you to plot a mathematical function without creating an explicit R function:

```
curve(sin(exp(4 * x)), n = 1000)
```

Here `x` is a pronoun. `x` doesn't represent a single concrete value, but is instead a placeholder that varies over the range of the plot. One way to implement `curve()` would be with `make_function()`:

```
curve2 <- function(expr, xlim = c(0, 1), n = 100,
                   env = parent.frame()) {
  f <- make_function(alist(x = ), substitute(expr), env)

  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)

  plot(x, y, type = "l", ylab = deparse(substitute(expr)))
}
```

Functions that use a pronoun are called anaphoric (`http://en.wikipedia.org/wiki/Anaphora_(linguistics)`) functions. They are used in Arc (`http://www.arcfn.com/doc/anaphoric.html`) (a lisp like language), Perl (`http://www.perlmonks.org/index.pl?node_id=666047`), and Clojure (`http://amalloy.hubpages.com/hub/Unhygenic-anaphoric-Clojure-macros-for-fun-and-profit`).

14.5.1 Exercises

1. How are `alist(a)` and `alist(a = )` different? Think about both the input and the output.

2. Read the documentation and source code for `pryr::partial()`. What does it do? How does it work? Read the documentation and source code for `pryr::unenclose()`. What does it do and how does it work?

3. The actual implementation of `curve()` looks more like

   ```
   curve3 <- function(expr, xlim = c(0, 1), n = 100,
                      env = parent.frame()) {
     env2 <- new.env(parent = env)
     env2$x <- seq(xlim[1], xlim[2], length = n)

     y <- eval(substitute(expr), env2)
     plot(env2$x, y, type = "l",
       ylab = deparse(substitute(expr)))
   }
   ```

   How does this approach differ from `curve2()` defined above?

14.6 Parsing and deparsing

Sometimes code is represented as a string, rather than as an expression. You can convert a string to an expression with `parse()`. `parse()` is the opposite of `deparse()`: it takes a character vector and returns an expression object. The primary use of `parse()` is parsing files of code to disk, so the first argument is a file path. Note that if you have code in a character vector, you need to use the `text` argument:

```
z <- quote(y <- x * 10)
deparse(z)
#> [1] "y <- x * 10"

parse(text = deparse(z))
#> expression(y <- x * 10)
```

Because there might be many top-level calls in a file, `parse()` doesn't return just a single expression. Instead, it returns an expression object, which is essentially a list of expressions:

```
exp <- parse(text = c("
  x <- 4
  x
  5
"))
length(exp)
#> [1] 3
typeof(exp)
#> [1] "expression"

exp[[1]]
#> x <- 4
exp[[2]]
#> x
```

You can create expression objects by hand with `expression()`, but I wouldn't recommend it. There's no need to learn about this esoteric data structure if you already know how to use expressions.

With `parse()` and `eval()`, it's possible to write a simple version of `source()`. We read in the file from disk, `parse()` it and then `eval()` each component in a specified environment. This version defaults to a new environment, so it doesn't affect existing objects. `source()` invisibly returns the result of the last expression in the file, so `simple_source()` does the same.

```
simple_source <- function(file, envir = new.env()) {
  stopifnot(file.exists(file))
  stopifnot(is.environment(envir))

  lines <- readLines(file, warn = FALSE)
  exprs <- parse(text = lines)

  n <- length(exprs)
  if (n == 0L) return(invisible())

  for (i in seq_len(n - 1)) {
    eval(exprs[i], envir)
  }
  invisible(eval(exprs[n], envir))
}
```

The real `source()` is considerably more complicated because it can `echo` input and output, and also has many additional settings to control behaviour.

14.6.1 Exercises

1. What are the differences between `quote()` and `expression()`?
2. Read the help for `deparse()` and construct a call that `deparse()` and `parse()` do not operate symmetrically on.
3. Compare and contrast `source()` and `sys.source()`.
4. Modify `simple_source()` so it returns the result of *every* expression, not just the last one.
5. The code generated by `simple_source()` lacks source references. Read the source code for `sys.source()` and the help for `srcfilecopy()`, then modify `simple_source()` to preserve source references. You can test your code by sourcing a function that contains a comment. If successful, when you look at the function, you'll see the comment and not just the source code.

14.7 Walking the AST with recursive functions

It's easy to modify a single call with `substitute()` or `pryr::modify_call()`. For more complicated tasks we need to work directly with the AST. The base `codetools` package provides some useful motivating examples of how we can do this:

- `findGlobals()` locates all global variables used by a function. This can be useful if you want to check that your function doesn't inadvertently rely on variables defined in their parent environment.
- `checkUsage()` checks for a range of common problems including unused local variables, unused parameters, and the use of partial argument matching.

To write functions like `findGlobals()` and `checkUsage()`, we'll need a new tool. Because expressions have a tree structure, using a recursive

function would be the natural choice. The key to doing that is getting the recursion right. This means making sure that you know what the base case is and figuring out how to combine the results from the recursive case. For calls, there are two base cases (atomic vectors and names) and two recursive cases (calls and pairlists). This means that a function for working with expressions will look like:

```
recurse_call <- function(x) {
  if (is.atomic(x)) {
    # Return a value
  } else if (is.name(x)) {
    # Return a value
  } else if (is.call(x)) {
    # Call recurse_call recursively
  } else if (is.pairlist(x)) {
    # Call recurse_call recursively
  } else {
    # User supplied incorrect input
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}
```

14.7.1 Finding F and T

We'll start simple with a function that determines whether a function uses the logical abbreviations `T` and `F`. Using `T` and `F` is generally considered to be poor coding practice, and is something that `R CMD check` will warn about. Let's first compare the AST for `T` vs. `TRUE`:

```
ast(TRUE)
#> \-  TRUE
ast(T)
#> \- `T
```

`TRUE` is parsed as a logical vector of length one, while `T` is parsed as a name. This tells us how to write our base cases for the recursive function: while an atomic vector will never be a logical abbreviation, a name might, so we'll need to test for both `T` and `F`. The recursive cases can be combined because they do the same thing in both cases: they recursively call `logical_abbr()` on each element of the object.

```
logical_abbr <- function(x) {
  if (is.atomic(x)) {
    FALSE
  } else if (is.name(x)) {
    identical(x, quote(T)) || identical(x, quote(F))
  } else if (is.call(x) || is.pairlist(x)) {
    for (i in seq_along(x)) {
      if (logical_abbr(x[[i]])) return(TRUE)
    }
    FALSE
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

logical_abbr(quote(TRUE))
#> [1] FALSE
logical_abbr(quote(T))
#> [1] TRUE
logical_abbr(quote(mean(x, na.rm = T)))
#> [1] TRUE
logical_abbr(quote(function(x, na.rm = T) FALSE))
#> [1] TRUE
```

14.7.2 Finding all variables created by assignment

`logical_abbr()` is very simple: it only returns a single `TRUE` or `FALSE`. The next task, listing all variables created by assignment, is a little more complicated. We'll start simply, and then make the function progressively more rigorous.

Again, we start by looking at the AST for assignment:

```
ast(x <- 10)
#> \- ()
#>   \- `<-
#>   \- `x
#>   \-  10
```

Assignment is a call where the first element is the name `<-`, the second is the object the name is assigned to, and the third is the value to be

assigned. This makes the base cases simple: constants and names don't create assignments, so they return `NULL`. The recursive cases aren't too hard either. We `lapply()` over pairlists and over calls to functions other than `<-`.

```
find_assign <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    NULL
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      x[[2]]
    } else {
      lapply(x, find_assign)
    }
  } else if (is.pairlist(x)) {
    lapply(x, find_assign)
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}
find_assign(quote(a <- 1))
#> a
find_assign(quote({
  a <- 1
  b <- 2
}))
#> [[1]]
#> NULL
#>
#> [[2]]
#> a
#>
#> [[3]]
#> b
```

This function works for these simple cases, but the output is rather verbose and includes some extraneous `NULL`s. Instead of returning a list, let's keep it simple and use a character vector. We'll also test it with two slightly more complicated examples:

```
find_assign2 <- function(x) {
```

```
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      as.character(x[[2]])
    } else {
      unlist(lapply(x, find_assign2))
    }
  } else if (is.pairlist(x)) {
    unlist(lapply(x, find_assign2))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

find_assign2(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b" "a"

find_assign2(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x"
```

This is better, but we have two problems: dealing with repeated names and neglecting assignments inside other assignments. The fix for the first problem is easy. We need to wrap `unique()` around the recursive case to remove duplicate assignments. The fix for the second problem is a bit more tricky. We also need to recurse when the call is to `<-`. `find_assign3()` implements both strategies:

```
find_assign3 <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      lhs <- as.character(x[[2]])
    } else {
      lhs <- character()
```

```
    }

    unique(c(lhs, unlist(lapply(x, find_assign3))))
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x, find_assign3)))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

find_assign3(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b"

find_assign3(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x" "y"
```

We also need to test subassignment:

```
find_assign3(quote({
  l <- list()
  l$a <- 5
  names(l) <- "b"
}))
#> [1] "l"     "$"     "a"     "names"
```

We only want assignment of the object itself, not assignment that modifies a property of the object. Drawing the tree for the quoted object will help us see what condition to test for. The second element of the call to `<-` should be a name, not another call.

```
ast(l$a <- 5)
#> \- ()
#>   \- `<-
#>   \- ()
#>     \- `$
```

```
#>     \- `l
#>     \- `a
#>   \-  5
ast(names(l) <- "b")
#> \- ()
#>   \- `<-
#>   \- ()
#>     \- `names
#>     \- `l
#>   \-  "b"
```

Now we have a complete version:

```
find_assign4 <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`)) && is.name(x[[2]])) {
      lhs <- as.character(x[[2]])
    } else {
      lhs <- character()
    }

    unique(c(lhs, unlist(lapply(x, find_assign4))))
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x, find_assign4)))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

find_assign4(quote({
  l <- list()
  l$a <- 5
  names(l) <- "b"
}))
#> [1] "l"
```

While the complete version of this function is quite complicated, it's important to remember we wrote it by working our way up by writing simple component parts.

14.7.3 Modifying the call tree

The next step up in complexity is returning a modified call tree, like what you get with `bquote()`. `bquote()` is a slightly more flexible form of quote: it allows you to optionally quote and unquote some parts of an expression (it's similar to the backtick operator in Lisp). Everything is quoted, *unless* it's encapsulated in `.()` in which case it's evaluated and the result is inserted:

```
a <- 1
b <- 3
bquote(a + b)
#> a + b
bquote(a + .(b))
#> a + 3
bquote(.(a) + .(b))
#> 1 + 3
bquote(.(a + b))
#> [1] 4
```

This provides a fairly easy way to control what gets evaluated and when. How does `bquote()` work? Below, I've rewritten `bquote()` to use the same style as our other functions: it expects input to be quoted already, and makes the base and recursive cases more explicit:

```
bquote2 <- function (x, where = parent.frame()) {
  if (is.atomic(x) || is.name(x)) {
    # Leave unchanged
    x
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(.))) {
      # Call to .(), so evaluate
      eval(x[[2]], where)
    } else {
      # Otherwise apply recursively, turning result back into call
      as.call(lapply(x, bquote2, where = where))
    }
  } else if (is.pairlist(x)) {
    as.pairlist(lapply(x, bquote2, where = where))
  } else {
    # User supplied incorrect input
    stop("Don't know how to handle type ", typeof(x),
```

```
      call. = FALSE)
  }
}

x <- 1
y <- 2
bquote2(quote(x == .(x)))
#> x == 1
bquote2(quote(function(x = .(x)) {
  x + .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

The main difference between this and the previous recursive functions is that after we process each element of calls and pairlists, we need to coerce them back to their original types.

Note that functions that modify the source tree are most useful for creating expressions that are used at run-time, rather than those that are saved back to the original source file. This is because all non-code information is lost:

```
bquote2(quote(function(x = .(x)) {
  # This is a comment
  x +  # funky spacing
    .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

These tools are somewhat similar to Lisp macros, as discussed in Programmer's Niche: Macros in R (`http://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf#page=10`) by Thomas Lumley. However, macros are run at compile-time, which doesn't have any meaning in R, and always return expressions. They're also somewhat like Lisp fexprs (`http://en.wikipedia.org/wiki/Fexpr`). A fexpr is a function where the arguments are not evaluated by default. The terms macro and fexpr are useful to know when looking for useful techniques from other languages.

14.7.4 Exercises

1. Why does `logical_abbr()` use a for loop instead of a functional like `lapply()`?

2. `logical_abbr()` works when given quoted objects, but doesn't work when given an existing function, as in the example below. Why not? How could you modify `logical_abbr()` to work with functions? Think about what components make up a function.

   ```
   f <- function(x = TRUE) {
     g(x + T)
   }
   logical_abbr(f)
   ```

3. Write a function called `ast_type()` that returns either "constant", "name", "call", or "pairlist". Rewrite `logical_abbr()`, `find_assign()`, and `bquote2()` to use this function with `switch()` instead of nested if statements.

4. Write a function that extracts all calls to a function. Compare your function to `pryr::fun_calls()`.

5. Write a wrapper around `bquote2()` that does non-standard evaluation so that you don't need to explicitly `quote()` the input.

6. Compare `bquote2()` to `bquote()`. There is a subtle bug in `bquote()`: it won't replace calls to functions with no arguments. Why?

   ```
   bquote(.(x)(), list(x = quote(f)))
   #> .(x)()
   bquote(.(x)(1), list(x = quote(f)))
   #> f(1)
   ```

7. Improve the base `recurse_call()` template to also work with lists of functions and expressions (e.g., as from `parse(path_to_file)`).

15

Domain specific languages

The combination of first class environments, lexical scoping, non-standard evaluation, and metaprogramming gives us a powerful toolkit for creating embedded domain specific languages (DSLs) in R. Embedded DSLs take advantage of a host language's parsing and execution framework, but adjust the semantics to make them more suitable for a specific task. DSLs are a very large topic, and this chapter will only scratch the surface, focussing on important implementation techniques rather than on how you might come up with the language in the first place. If you're interested in learning more, I highly recommend *Domain Specific Languages* (`http://amzn.com/0321712943?tag=devtools-20`) by Martin Fowler. It discusses many options for creating a DSL and provides many examples of different languages.

R's most popular DSL is the formula specification, which provides a succinct way of describing the relationship between predictors and the response in a model. Other examples include ggplot2 (for visualisation) and plyr (for data manipulation). Another package that makes extensive use of these ideas is dplyr, which provides `translate_sql()` to convert R expressions into SQL:

```
library(dplyr)
translate_sql(sin(x) + tan(y))
#> <SQL> SIN("x") + TAN("y")
translate_sql(x < 5 & !(y >= 5))
#> <SQL> "x" < 5.0 AND NOT(("y" >= 5.0))
translate_sql(first %like% "Had*")
#> <SQL> "first" LIKE 'Had*'
translate_sql(first %in% c("John", "Roger", "Robert"))
#> <SQL> "first" IN ('John', 'Roger', 'Robert')
translate_sql(like == 7)
#> <SQL> "like" = 7.0
```

This chapter will develop two simple, but useful DSLs: one to generate

HTML, and the other to turn mathematical expressions expressed in R code into LaTeX.

Prerequisites

This chapter together pulls together many techniques discussed elsewhere in the book. In particular, you'll need to understand environments, functionals, non-standard evaluation, and metaprogramming.

15.1 HTML

HTML (hypertext markup language) is the language that underlies the majority of the web. It's a special case of SGML (standard generalised markup language), and it's similar but not identical to XML (extensible markup language). HTML looks like this:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Even if you've never looked at HTML before, you can still see that the key component of its coding structure is tags, `<tag></tag>`. Tags can be contained inside other tags and intermingled with text. Generally, HTML ignores whitespaces (a sequence of whitespace is equivalent to a single space) so you could put the previous example on a single line and it would still display the same in a browser:

```
<body><h1 id='first'>A heading</h1><p>Some text &amp; <b>some bold
text.</b></p><img src='myimg.png' width='100' height='100' />
</body>
```

However, like R code, you usually want to indent HTML to make the structure more obvious.

There are over 100 HTML tags. But to illustrate HTML, we're going to focus on just a few:

- `<body>`: the top-level tag that all content is enclosed within
- `<h1>`: creates a heading-1, the top level heading
- `<p>`: creates a paragraph
- `<b>`: emboldens text
- `<img>`: embeds an image

(You probably guessed what these did already!)

Tags can also have named attributes. They look like `<tag a="a" b="b"></tag>`. Tag values should always be enclosed in either single or double quotes. Two important attributes used with just about every tag are `id` and `class`. These are used in conjunction with CSS (cascading style sheets) in order to control the style of the document.

Some tags, like `<img>`, can't have any content. These are called **void tags** and have a slightly different syntax. Instead of writing `<img></img>`, you write `<img />`. Since they have no content, attributes are more important. In fact, `img` has three that are used for almost every image: `src` (where the image lives), `width`, and `height`.

Because `<` and `>` have special meanings in HTML, you can't write them directly. Instead you have to use the HTML escapes: `&gt;` and `&lt;`. And, since those escapes use `&`, if you want a literal ampersand you have to escape with `&amp;`.

15.1.1 Goal

Our goal is to make it easy to generate HTML from R. To give a concrete example, we want to generate the following HTML with code that looks as similar to the HTML as possible.

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp;amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

To do so, we will work our way up to the following DSL:

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
```

Note that the nesting of function calls is the same as the nesting of tags: unnamed arguments become the content of the tag, and named arguments become their attributes. Because tags and text are clearly distinct in this API, we can automatically escape `&` and other special characters.

15.1.2 Escaping

Escaping is so fundamental to DSLs that it'll be our first topic. To create a way of escaping characters, we need to give "&" a special meaning without ending up double-escaping. The easiest way to do this is to create an S3 class that distinguishes between regular text (that needs escaping) and HTML (that doesn't).

```
html <- function(x) structure(x, class = "html")
print.html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

We then write an escape method that leaves HTML unchanged and escapes the special characters (`&`, `<`, `>`) for ordinary text. We also add a list method for convenience.

```
escape <- function(x) UseMethod("escape")
escape.html <- function(x) x
escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}
escape.list <- function(x) {
  lapply(x, escape)
}

# Now we check that it works
escape("This is some text.")
#> <HTML> This is some text.
escape("x > 1 & y < 2")
```

```
#> <HTML> x &gt; 1 &amp; y &lt; 2

# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))
#> <HTML> <hr />
```

Escaping is an important component for many DSLs.

15.1.3 Basic tag functions

Next, we'll write a few simple tag functions and then figure out how to generalise this function to cover all possible HTML tags. Let's start with `<p>`. HTML tags can have both attributes (e.g., id or class) and children (like `<b>` or `<i>`). We need some way of separating these in the function call. Given that attributes are named values and children don't have names, it seems natural to separate using named arguments from unnamed ones. For example, a call to `p()` might look like:

```
p("Some text.", b("some bold text"), class = "mypara")
```

We could list all the possible attributes of the `<p>` tag in the function definition. However, that's hard not only because there are many attributes, but also because it's possible to use custom attributes (`http://html5doctor.com/html5-custom-data-attributes/`). Instead, we'll just use `...` and separate the components based on whether or not they are named. To do this correctly, we need to be aware of an inconsistency in `names()`:

```
names(c(a = 1, b = 2))
#> [1] "a" "b"
names(c(a = 1, 2))
#> [1] "a" ""
names(c(1, 2))
#> NULL
```

With this in mind, we create two helper functions to extract the named and unnamed components of a vector:

```
named <- function(x) {
  if (is.null(names(x))) return(NULL)
  x[names(x) != ""]
}
unnamed <- function(x) {
  if (is.null(names(x))) return(x)
  x[names(x) == ""]
}
```

We can now create our `p()` function. Notice that there's one new function here: `html_attributes()`. It uses a list of name-value pairs to create the correct specification of HTML attributes. It's a little complicated (in part, because it deals with some idiosyncracies of HTML that I haven't mentioned.). However, because it's not that important and doesn't introduce any new ideas, I won't discuss it here (you can find the source online).

```
source("dsl-html-attributes.r", local = TRUE)
p <- function(...) {
  args <- list(...)
  attribs <- html_attributes(named(args))
  children <- unlist(escape(unnamed(args)))

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
  ))
}

p("Some text")
#> <HTML> <p>Some text</p>
p("Some text", id = "myid")
#> <HTML> <p id = 'myid'>Some text</p>
p("Some text", image = NULL)
#> <HTML> <p image>Some text</p>
p("Some text", class = "important", "data-value" = 10)
#> <HTML> <p class = 'important' data-value = '10'>Some
#> text</p>
```

15.1.4 Tag functions

With this definition of `p()`, it's pretty easy to see how we can apply this approach to different tags: we just need to replace `"p"` with a variable. We'll use a closure to make it easy to generate a tag function given a tag name:

```
tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    attribs <- html_attributes(named(args))
    children <- unlist(escape(unnamed(args)))

    html(paste0(
      "<", tag, attribs, ">",
      paste(children, collapse = ""),
      "</", tag, ">"
    ))
  }
}
```

(We're forcing the evaluation of `tag` with the expectation that we'll be calling this function from a loop. This will help to avoid potential bugs caused by lazy evaluation.)

Now we can run our earlier example:

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text.", b("Some bold text"), i("Some italic text"),
  class = "mypara")
#> <HTML> <p class = 'mypara'>Some text.<b>Some bold
#> text</b><i>Some italic text</i></p>
```

Before we continue writing functions for every possible HTML tag, we need to create a variant of `tag()` for void tags. It can be very similar to `tag()`, but if there are any unnamed tags, it needs to throw an error. Also note that the tag itself will look slightly different:

```
void_tag <- function(tag) {
  force(tag)
```