Hadley Wickham

Advanced R

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

Contents

Welcome

This is the website for work-in-progress 2nd edition of **"Advanced R (`http://amzn.com/1466586966?tag=devtools-20`)"**, a book in Chapman & Hall's R Series. The book is designed primarily for R users who want to improve their programming skills and understanding of the language. It should also be useful for programmers coming to R from other languages, as it explains some of R's quirks and shows how some parts that seem horrible do have a positive side.

This edition is a work in progress. If you're looking for the electronic version of the 1st edition, you can find it online at `http://adv-r.had.co.nz/`.

Other books

You may also be interested in:

- **"R for Data Science (`http://r4ds.had.co.nz/`)"** which introduces you to R as a tool for doing data science, focussing on a consistent set of packages known as the tidyverse.
- **"R Packages (`http://r-pkgs.had.co.nz/`)"** which teaches you how to make the most of R's fantastic package system.

0

Preface

Welcome to the work-in-progress 2nd edition of **Advanced R**. This preface describes the major changes that I have made to the book.

The 2nd edition has been published in colour, which as well as improving the syntax highlighting of the code chunks, has considerably increased the scope for helpful diagrams. I have taken advantage of this and included many more diagrams throughout the book.

0.1 rlang

A big change since the first edition of the book is the creation of the rlang (`http://rlang.r-lib.org`) package, written primarily by Lionel Henry. The goal of this package is to provide a clean interface to low-level data structures and operations. I use this package in favour of base R because I believe it makes easier to understand how the R language works. Instead of struggling with the incidentals of functions that evolved organically over many years, the more consistent rlang API makes it easier to focus on the big ideas.

In each section, I'll briefly outline the base R equivalents to rlang code. But if you want to see the purest base R expression of these ideas, I recommend reading the first edition of the book, which you can find online at `http://adv-r.had.co.nz`.

Overall, rlang is still a work in progress, and much of the API continues to mature. However, the code used in this book is part of the rlang's testing process and will continue to work in the future. You can also see our confidence in the stability of rlang functions with the lifecycle badges at the documentation.

0.2 Foundations

- Environments: more pictures. Much improved discussion of frames and how they relate to the call stack.
- New chapter on "Names and values" that helps you form a better mental of `<-`, and to better understand when R makes copies of existing data structures. Understanding the distinction between names and values is important for functional programming, and understanding when R makes copies is critical for accurate performance predictions.
- Vectors (previously data structures) has been rewritten with more diagrams to focus on vector types. More information about other important S3 vectors, and information about tibbles, a modern re-imagining of data frames.
- Exceptions and debugging has been split into two chapters, "debugging" and "conditions". The contents of conditions has been expanded. The section of defensive programming has been removed, because discussing type stability is more natural in the context of functional programming, and programming with NSE is not the challenge it once was (now that tidy evaluation exists).

0.3 Programming paradigms

After foundations, the book is now organised around the three most important programming paradigms in R:

- Functional programming has been updated to focus on the tools provided by the purrr package. The greatear consistency in the purrr package makes it possible to focus more on the underlying ideas without being distracted by incidental details. Divided more cleanly into functionals, function factories, and function operators. Greater focus on what time has shown to be important in practice. Less math + stat, and more data science.
- Object oriented programming (OOP) now forms a major section of the book with individual chapters on base types, S3, S4, R6, and the tradeoffs between the systems.
- Metaprogramming, formerly computing on the language, describes the suite of tools that you can use to generate code with code. Compared

to the first edition has been substantially expanded (from three chapters to five) and reorganised. More diagrams.

0.4 Techniques

Final section discusses programming techniques, including both debugging, profiling, improving performance, and connecting R and C++.

0.5 Removals

- Chapter of base R vocabulary was removed.
- The style guide has moved to `http://style.tidyverse.org/`. It is now paired with the styler (`http://styler.r-lib.org/`) package which can automatically apply many of the rules.
- R's C interface moving to the work-in-progress `https://github.com/hadley/r-internals`
- Memory chapter either integrated in names and values, or removed because it's excessively technical and not that important to understand (unless you're working with C code in which case it belongs in internals).

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 like 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`), RStudio Community (`https://community.rstudio.com/`), 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 (https://jumpingrivers.github.io/meetingsR/).

- 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()`).

This books works the narrow line between being a reference book (primarily used for lookup), and being linearly readable. This involves some tradeoffs, because it's difficult to linearise material while still keeping related materials together, whereas some concepts are much easier to explain if you're already familiar with specific technically vocabulary. I've tried to use footnotes and cross-references to make sure you can still make sense even if you just dip your toes in the occassional chapter.

1.2 Related work

Tidyverse + R4DS

R packages

1.3 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.

1.4 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.

1.5 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* (`https://mitpress.mit.edu/sites/default/files/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.

1.6 Getting help

Currently, there are three main venues to get help when you're stuck and can't figure out what's causing the problem: RStudio Community (https://community.rstudio.com/), stackoverflow (http://stackoverflow.com) and the R-help mailing list. You can get fantastic help in each venue, 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. The `reprex` (https://reprex.tidyverse.org/) package can help you create a **repr**oducible **ex**ample that will can easily be run by people trying to help you. There are several resources available (https://community.rstudio.com/t/faq-whats-a-reproducible-example-reprex-and-how-do-i-do-one/5219) to help you create a successful `reprex`.
- 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.

1.7 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 Wolen (@aaronwolen), @absolutelyNoWarranty, Adam Hunt (@adamphunt), @agrabovsky, Alexander Grueneberg (@agrueneberg), Anthony Damico (@ajdamico), James Manton (@ajdm), Aaron Schumacher (@ajschumacher), Alan Dipert (@alandipert), Alex Brown (@alexbbrown), @alexperrone, Alex Whitworth (@alexWhitworth), Alexandros Kokkalis (@alko989), @amarchin, Amelia McNamara (@AmeliaMN), Bryce Mecum (@amoeba), Andrew Laucius (@andrewla), Andrew Bray (@andrewpbray), Andrie de Vries (@andrie), @aranlunzer, Ari Lamstein (@arilamstein), @asnr, Andy Teucher (@ateucher), Albert Vilella (@avilella), baptiste (@baptiste), Brian G. Barkley (@BarkleyBG), Mara Averick (@batpigandme), Barbara Borges Ribeiro (@bborgesr), Brandon Greenwell (@bgreenwell), Brandon Hurr (@bhive01), Jason Knight (@binarybana), Brett Klamer (@bklamer), Jesse Anderson (@blindjesse), Brian Mayer (@blmayer), Benjamin L. Moore (@blmoore), Brian Diggs (@BrianDiggs), Brian S. Yandell (@byandell), @carey1024, Chip Hogg (@chiphogg), Chris Muir (@ChrisMuir), Christopher Gandrud (@christophergandrud), Clay Ford (@clayford), Colin Fay (@ColinFay), @cortinah, Cameron Plouffe (@cplouffe), Carson Sievert (@cpsievert), Craig Citro (@craigcitro), Craig Grabowski (@craiggrabowski), Christopher Roach (@croach), Peter Meilstrup (@crowding), Crt Ahlin (@crtahlin), Carlos Scheidegger (@cscheid), Colin Gillespie (@csgillespie), Christopher Brown (@ctbrown), Davor Cubranic (@cubranic), Darren Cusanovich (@cusanovich), Christian G. Warden (@cwarden), Charlotte Wickham (@cwickham), Dean Attali (@daattali), Dan Sullivan (@dan87134), Daniel Barnett (@daniel-barnett), Kenny Darrell (@darrkj), Tracy Nance (@datapixie), Dave Childers (@davechilders), David Rubinger (@davidrubinger), David Chudzicki (@dchudz), Daisuke ICHIKAWA (@dichika), david kahle (@dkahle), David LeBauer (@dlebauer), David Schweizer (@dlschweizer), David Montaner (@dmontaner), Zhuoer Dong (@dongzhuoer), Doug Mitarotonda (@dougmitarotonda), Jonathan Hill (@Dripdrop12), Julian During (@duju211), @duncanwadsworth, @eaurele, Dirk Eddelbuettel (@eddelbuettel), @EdFineOKL, Edwin Thoen (@EdwinTh), Ethan Heinzen (@eheinzen), @eijoac, Joel Schwartz (@eipi10), Eric Ronald Legrand (@elegrand), Ellis Valentiner (@ellisvalentiner), Emil Hvit-

feldt (@EmilHvitfeldt), Emil Rehnberg (@EmilRehnberg), Daniel Lee (@erget), Eric C. Anderson (@eriqande), Enrico Spinielli (@espinielli), @etb, David Hajage (@eusebe), Fabian Scheipl (@fabian-s), @flammy0530, François Michonneau (@fmichonneau), Francois Pepin (@fpepin), Frank Farach (@frankfarach), @freezby, Frans van Dunné (@FvD), @fyears, @gagnagaman, Garrett Grolemund (@garrettgman), Gavin Simpson (@gavinsimpson), @gezakiss7, @gggtest, Gökçen Eraslan (@gokceneraslan), Georg Russ (@gr650), @grasshoppermouse, Gregor Thomas (@gregorp), Garrett See (@gsee), Ari Friedman (@gsk3), Gunnlaugur Thor Briem (@gthb), Hadley Wickham (@hadley), Hamed (@hamedbh), Harley Day (@harleyday), @hassaad85, @helmingstay, Henning (@henningsway), Henrik Bengtsson (@HenrikBengtsson), Ching Boon (@hoscb), Iain Dillingham (@iaindillingham), @IanKopacka, Ian Lyttle (@ijlyttle), Ilan Man (@ilanman), Imanuel Costigan (@imanuelcostigan), Thomas Bürli (@initdch), Os Keyes (@Ironholds), @irudnyts, i (@isomorphisms), Irene Steves (@isteves), Jan Gleixner (@jan-glx), Jason Asher (@jasonasher), Jason Davies (@jasondavies), Chris (@jastingo), jcborras (@jcborras), John Blischak (@jdblischak), @jeharmse, Lukas Burk (@jemus42), Jennifer (Jenny) Bryan (@jennybc), Justin Jent (@jentjr), Jeston (@JestonBlu), Jim Hester (@jimhester), @JimInNashville, @jimmyliu2017, Jim Vine (@jimvine), Jinlong Yang (@jinlong25), J.J. Allaire (@jjallaire), @JMHay, Jochen Van de Velde (@jochenvdv), Johann Hibschman (@johannh), John Baumgartner (@johnbaums), John Horton (@johnjosephhorton), @johnthomas12, Jon Calder (@jonmcalder), Jon Harmon (@jonthegeek), Julia Gustavsen (@jooolia), JorneBiccler (@JorneBiccler), Jeffrey Arnold (@jrnold), Joyce Robbins (@jtr13), Juan Manuel Truppia (@juancentro), Kevin Markham (@justmarkham), john verzani (@jverzani), Michael Kane (@kaneplusplus), Bart Kastermans (@kasterma), Kevin D'Auria (@kdauria), Karandeep Singh (@kdpsingh), Ken Williams (@kenahoo), Kendon Bell (@kendonB), Kent Johnson (@kent37), Kevin Ushey (@kevinushey), (@kfeng123), Karl Forner (@kforner), Kirill Sevastyanenko (@kirillseva), Brian Knaus (@knausb), Kirill Müller (@krlmlr), Kriti Sen Sharma (@ksens), Kevin Wright (@kwstat), `suo.lawrence.liu@gmail.com` (`mailto:suo.lawrence.liu@gmail.com`) (@Lawrence-Liu), @ldfmrails, Rachel Severson (@leighseverson), Laurent Gatto (@lgatto), C. Jason Liang (@liangcj), Steve Lianoglou (@lianos), @lindbrook, Lingbing Feng (@Lingbing), Marcel Ramos (@LiNk-NY), Zhongpeng Lin (@linzhp), Lionel Henry (@lionel-), myq (@lrcg), Luke W Johnston (@lwjohnst86), Kevin Lynagh (@lynaghk), Malcolm Barrett (@malcolmbarrett), @mannyishere, Matt (@mattbaggott), Matthew Grogan (@mattgrogan), @matthewhillary, Matthieu Gomez (@matthieugomez), Matt Malin (@mattmalin), Mauro Lepore (@maurolepore), Max Ghenis (@MaxGhenis), Maximilian Held (@maxheld83), Michal Bojanowski (@mbojan), Mark Rosenstein (@mbrmbr), Michael Sumner (@mdsumner), Jun Mei (@meijun), merkliopas (@merkliopas), mfrasco (@mfrasco), Michael Bach (@michaelbach), Michael Bishop (@MichaelMBishop), Michael Buckley (@michaelmikebuckley), Michael Quinn (@michaelquinn32), @miguel-

morin, Michael (@mikekaminsky), Mine Cetinkaya-Rundel (@mine-cetinkaya-rundel), @mjsduncan, Mamoun Benghezal (@MoBeng), Matt Pettis (@mpettis), Martin Morgan (@mtmorgan), Guy Dawson (@Mullefa), Nacho Caballero (@nachocab), Natalya Rapstine (@natalya-patrikeeva), Nick Carchedi (@ncarchedi), Noah Greifer (@ngreifer), Nicholas Vasile (@nickv9), Nikos Ignatiadis (@nignatiadis), Xavier Laviron (@norival), Nick Pullen (@nstjhp), Oge Nnadi (@ogennadi), Oliver Paisley (@oliverpaisley), Pariksheet Nanda (@omsai), Øystein Sørensen (@osorensen), Paul (@otepoti), Otho Mantegazza (@othomantegazza), Dewey Dunnington (@paleolimbot), Parker Abercrombie (@parkerabercrombie), Patrick Hausmann (@patperu), Patrick Miller (@patr1ckm), Patrick Werkmeister (@Patrick01), @paulponcet, @pdb61, Tom Crockett (@pelotom), @pengyu, Jeremiah (@perryjer1), Peter Hickey (@PeteHaitch), Phil Chalmers (@philchalmers), Jose Antonio Magaña Mesa (@picarus), Pierre Casadebaig (@picasa), Antonio Piccolboni (@piccolbo), Pierre Roudier (@pierreroudier), Poor Yorick (@pooryorick), Marie-Helene Burle (@prosoitos), Peter Schulam (@pschulam), John (@quantbo), Quyu Kong (@qykong), Ramiro Magno (@ramiromagno), Ramnath Vaidyanathan (@ramnathv), Kun Ren (@renkun-ken), Richard Reeve (@richardreeve), Richard Cotton (@richierocks), Robert M Flight (@rmflight), R. Mark Sharp (@rmsharp), Robert Krzyzanowski (@robertzk), @robiRagan, Romain François (@romain-francois), Ross Holmberg (@rossholmberg), Ricardo Pietrobon (@rpietro), @rrunner, Ryan Walker (@rtwalker), @rubenfcasal, Rob Weyant (@rweyant), Rumen Zarev (@rzarev), Nan Wang (@sailingwave), @sbgraves237, Scott Kostyshak (@scottkosty), Scott Leishman (@scttl), Sean Hughes (@seaaan), Sean Anderson (@seananderson), Sean Carmody (@seancarmody), Sebastian (@sebastian-c), Matthew Sedaghatfar (@sedaghatfar), @see24, Sven E. Templer (@setempler), @sflippl, @shabbybanks, Steven Pav (@shabbychef), Shannon Rush (@shannonrush), S'busiso Mkhondwane (@sibusiso16), Sigfried Gold (@Sigfried), Simon O'Hanlon (@simonohanlon101), Simon Potter (@sjp), Steve (@SplashDance), Scott Ritchie (@sritchie73), Tim Cole (@statist7), @ste-fan, @stephens999, Steve Walker (@stevencarlislewalker), Stefan Widgren (@stewid), Homer Strong (@strongh), Dirk (@surmann), Sebastien Vigneau (@svigneau), Scott Warchal (@Swarchal), Steven Nydick (@swnydick), Taekyun Kim (@taekyunk), Tal Galili (@talgalili), @Tazinho, Tom B (@tbuckl), @tdenes, @thomasherbig, Thomas (@thomaskern), Thomas Lin Pedersen (@thomasp85), Thomas Zumbrunn (@thomaszumbrunn), Tim Waterhouse (@timwaterhouse), TJ Mahr (@tjmahr), Anton Antonov (@tonytonov), Ben Torvaney (@Torvaney), Jeff Allen (@trestletech), Terence Teo (@tteo), Tim Triche, Jr. (@ttriche), @tyhenkaline, Tyler Ritchie (@tylerritchie), Varun Agrawal (@varun729), Vijay Barve (@vijaybarve), Victor (@vkryukov), Vaidotas Zemlys-Balevičius (@vzemlys), Winston Chang (@wch), Linda Chin (@wchi144), Welliton Souza (@Welliton309), Gregg Whitworth (@whitwort), Will Beasley (@wibeasley), William R Bauer (@WilCrofter), William Doane (@WilDoane), Sean Wilkinson (@wilkinson), Christof Winter (@winterschlaefer), Bill Carver (@wmc3), Wolfgang Huber

(@wolfganghuber), Krishna Sankar (@xsankar), Yihui Xie (@yihui), yang (@yiluheihei), Yoni Ben-Meshulam (@yoni), @yuchouchen, @zachcp, @zackham, Edward Cho (@zerokarmaleft), Albert Zhao (@zxzb).

1.8 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., `https://adv-r.hadley.nz/`, you can easily copy and paste examples into R. Output comments look like `#>` to distinguish them from regular comments.

Many examples use random numbers. These are made reproducible by `set.seed(1014)` which is run at the start of each chapter.

1.9 Colophon

This book was written in bookdown (`http://bookdown.org/`) inside RStudio (`http://www.rstudio.com/ide/`). The website (`https://adv-r.hadley.nz/`) is hosted with netlify (`http://netlify.com/`), and automatically updated after every commit by travis-ci (`https://travis-ci.org/`). The complete source is available from github (`https://github.com/hadley/adv-r`).

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

setting	value
version	R version 3.5.1 (2018-07-02)
os	macOS High Sierra 10.13.6
system	x86_64, darwin15.6.0
ui	X11
language	(EN)
collate	en_US.UTF-8
tz	America/New_York
date	2018-09-02

```
#> Warning in fun(libname, pkgname): couldn't connect to display ":0"
```

package	version	source
assertthat	0.2.0	CRAN (R 3.5.0)
backports	1.1.2	CRAN (R 3.5.0)
base64enc	0.1-3	CRAN (R 3.5.0)
BH	1.66.0-1	CRAN (R 3.5.0)
bindr	0.1.1	CRAN (R 3.5.0)
bindrcpp	0.2.2	CRAN (R 3.5.0)
bit	1.1-14	CRAN (R 3.5.0)
bit64	0.9-7	CRAN (R 3.5.0)
bitops	1.0-6	CRAN (R 3.5.0)
blob	1.1.1	CRAN (R 3.5.0)
bookdown	0.7	CRAN (R 3.5.0)
boot	1.3-20	CRAN (R 3.5.1)
caTools	1.17.1.1	CRAN (R 3.5.0)
class	7.3-14	CRAN (R 3.5.1)
cli	1.0.0	CRAN (R 3.5.0)
clisymbols	1.2.0	CRAN (R 3.5.0)
clusrank	0.6-2	CRAN (R 3.5.0)
cluster	2.0.7-1	CRAN (R 3.5.1)
codetools	0.2-15	CRAN (R 3.5.1)
colorspace	1.3-2	CRAN (R 3.5.0)
crayon	1.3.4	CRAN (R 3.5.0)
DBI	1.0.0	CRAN (R 3.5.0)
dbplyr	1.2.2	CRAN (R 3.5.0)
dichromat	2.0-0	CRAN (R 3.5.0)
digest	0.6.16	CRAN (R 3.5.0)
dplyr	0.7.6	CRAN (R 3.5.1)
emo	0.0.0.9000	Github (hadley/emo\@02a5206)
EnvStats	2.3.1	CRAN (R 3.5.0)
eva	0.2.4	CRAN (R 3.5.0)
evaluate	0.11	CRAN (R 3.5.0)
fansi	0.3.0	CRAN (R 3.5.0)
foreign	0.8-70	CRAN (R 3.5.1)
ggplot2	3.0.0	CRAN (R 3.5.0)
glue	1.3.0	Github (tidyverse/glue\@4e74901)
gtable	0.2.0	CRAN (R 3.5.0)
highr	0.7	CRAN (R 3.5.0)
hms	0.4.2	CRAN (R 3.5.0)
htmltools	0.3.6	CRAN (R 3.5.0)
httpuv	1.4.5	CRAN (R 3.5.0)
jsonlite	1.5	CRAN (R 3.5.0)
KernSmooth	2.23-15	CRAN (R 3.5.1)
knitr	1.20	CRAN (R 3.5.0)
labeling	0.3	CRAN (R 3.5.0)
later	0.7.4	CRAN (R 3.5.0)
lattice	0.20-35	CRAN (R 3.5.1)
lazyeval	0.2.1	CRAN (R 3.5.0)
lineprof	0.1.9001	Github (hadley/lineprof\@972e71d)
lobstr	0.0.0.9000	Github (r-lib/lobstr\@530db70)
lubridate	1.7.4	CRAN (R 3.5.0)
magrittr	1.5	CRAN (R 3.5.0)
markdown	0.8	CRAN (R 3.5.0)
MASS	7.3-50	CRAN (R 3.5.0)

```
ruler()
#> ----+----1----+----2----+----3----+----4----+----5----+----6----+----
#> 12345678901234567890123456789012345678901234567890123456789012345678
```

Part I

Foundations

Introduction

To start your journey in mastering R, the following six chapters will help you learn what I consider to be the foundational components of R. I expect that you're already seen many of these pieces before, but you probably have not studied them deeply. To help check your existing knowledge, each chapter starts with a quiz; if you get all the questions right, feel free to skip to the next chapter!

1. In Chapter **??**, you'll learn about one of the most important distinctions you haven't previously needed to grapple with: the difference between an object and its name. Improving your mental model here will help you make better predictions about when R copies data and hence which basic operations are cheap and which are expensive.

2. Every day you've used R, you've used vectors, so Chapter **??** will dive into the details, helping you learn how the different types of vector fit together. You'll also learn about attributes, which allow you to store arbitrary metadata, and form the basis for two of R's object oriented programming toolkits

3. To write concise and performance R code it is important to fully appreciate the power of subsetting with `[`, `[[` and `$`, as described in Chapter **??**. Understanding the fundamental components of subsetting will allow you to solve new problems by combining the building blocks in novel ways.

4. Functions are the most important building block of R code, and in Chapter **??**, you'll learn exactly how they work, including the **scoping** rules, which govern how R looks up values from names. You'll also learn more of the details behind R's lazy evaluation, and how you can control what happens when you exit a function.

5. In Chapter **??**, you'll learn about a data structure that is crucial for understanding how R works, but quite unimportant for data analysis: the environment. Environments are the data structure that binds names to values, and they power tools like package namespaces. Unlike most programming languages, environments in R are "first class" which means that you can manipulate them just like other objects.

6. Chapter **??** concludes this section of the book with a discussion of "conditions", the umbrella term used to describe errors, warnings, and messages. You've certainly encountered these before, so in this chapter you learn how to signal them appropriately in your own functions, and how to handle them when signalled elsewhere.

2

Names and values

2.1 Introduction

In R, it is important to understand the distinction between an object and its name. A correct mental model is important because it will help you:

- More accurately predict performance and memory usage of R code.
- Write faster code because accidental copies are a major cause of slow code.
- Better understand R's functional programming tools.

The goal of this chapter is to help you understand the distinction between names and values, and when R will copy an object.

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 **??**.

1. Given the following data frame, how do I create a new column called "3" that contains the sum of `1` and `2`? You may only use `$`, not `[[`. What makes `1`, `2`, and `3` challenging as variable names?

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)
```

2. In the following code, how much memory does `y` occupy?

```
x <- runif(1e6)
y <- list(x, x, x)
```

3. On which line does `a` get copied in the following example?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

Outline

- Section **??** introduces you to the distinction between names and values, and discusses how `<-` creates a binding, or reference, between a name and a value.
- Section **??** describes when R makes a copy; whenever you modify vector, you're almost always actually create a new, modified vector. You'll learn how to use `tracemem()` to figure out when a copy actually occurs, and then explore the implications as they apply to function calls, lists, data frames, and character vectors.
- Section **??** explores the implications of the previous two sections on how much memory an object occupies. You'll learn to use `lobstr::obj_size()` as your intuition may be profoundly wrong, and the base `object.size()` is unfortunately inaccurate.
- Section **??** describes the two important exceptions to copy-on-modify: values with a single name, and environments. In these two special cases, objects are actually modified in place.
- Section **??** closes out the chapter with a discussion of the garbage collector, which frees up memory used by objects that are no longer referenced by a name.

Prerequisites

We'll use the development version of lobstr (`https://github.com/r-lib/lobstr`) to dig into the internal representation of R objects.

```
# devtools::install_github("r-lib/lobstr")
library(lobstr)
```

Sources

The details of R's memory management are not documented in a single place. Much of the information in this chapter was gleaned from a close reading of the documentation (particularly `?Memory` and `?gc`),

the memory profiling (http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use) section of "Writing R extensions" (?), and the SEXPs (http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs) section of "R internals" (?). The rest I figured out by reading the C source code, performing small experiments, and asking questions on R-devel. Any mistakes are entirely mine.

2.2 Binding basics

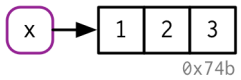
Take this code:

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

It's easy to read it as: "create an object named 'x', containing the values 1, 2, and 3". Unfortunately, that's a simplification that will lead to you make inaccurate predictions about what R is actually doing behind the scenes. It's more accurate to think about this code as doing two things:

- Creating an object, a vector of values, `c(1, 2, 3)`.
- Binding the object to a name, `x`.

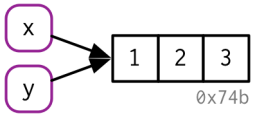
Note that the object, or value, doesn't have a name; it's the name that has a value. To make that distinction more clear, I'll draw diagrams like this:



The name, `x`, is drawn with a rounded rectangle, and it has an arrow that points to, binds, or references, the value, the vector `1:3`. Note that the arrow points in opposite direction to the assignment arrow: `<-` creates a binding from the name on the left-hand side to the object on the right-hand side.

You can think of a name as a reference to a value. For example, if you run this code, you don't get another copy of the value `1:3`, you get another binding to the existing object:

```
y <- x
```



You might have noticed the value `1:3` has a label: `0x74b`. While the vector doesn't have a name, I'll occasionally need to refer to objects independently of their bindings. To make that possible, I'll label values with a unique identifier. These unique identifers have a special form that looks like the object's memory "address", i.e. the location in memory in which the object is stored. It doesn't make sense to use the actual memory address because that changes every time the code is run.

You can access the address of an object with `lobstr::obj_addr()`. This allows us to see that `x` and `y` both point to the same location in memory:

```
obj_addr(x)
#> [1] "0x7ffb275183f8"
obj_addr(y)
#> [1] "0x7ffb275183f8"
```

These identifiers are long, and change every time you restart R.

It takes some time to get your head around the distinction between names and values, but it's really helpful for functional programming when you start to work with functions that have different names in different contexts.

2.2.1 Non-syntactic names

R has strict rules about what constitutes a valid name. A **syntactic** name must consist of letters[1], digits, . and _, and can't begin with _ or a digit. Additionally, it can not be one of a list of **reserved words** like `TRUE`, `NULL`, `if`, and `function` (see the complete list in `?Reserved`). Names that don't follow these rules are called **non-syntactic** names, and if you try to use them, you'll get an error:

```
_abc <- 1
#> Error: unexpected input in "_"

if <- 10
#> Error: unexpected assignment in "if <-"
```

[1]Surprisingly, what constitutes a letter is determined by your current locale. That means that the syntax of R code actually differs from computer to computer, and it's possible for a file that works on one computer to not even parse on another!

It's possible to override the usual rules and use a name with any sequence of characters by surrounding the name with backticks:

```
`_abc` <- 1
`_abc`
#> [1] 1

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

Typically, you won't deliberately create such crazy names. Instead, you need to understand them because you'll be subjected to the crazy names created by others. This happens most commonly when you load data that has been created outside of R.

You *can* create non-syntactic bindings using single or double quotes (e.g. `"_abc" <- 1`) instead of backticks, but you shouldn't, because you'll have to use a different syntax to retrieve the values. The ability to use strings on the left hand side of the assignment arrow is a historical artefact, used before R supported backticks.

2.2.2 Exercises

1. Explain the relationship between `a`, `b`, `c` and `d` in the following code:

   ```
   a <- 1:10
   b <- a
   c <- b
   d <- 1:10
   ```

2. The following code accesses the mean function in multiple different ways. Do they all point to the same underlying function object? Verify with `lobstr::obj_addr()`.

   ```
   mean
   base::mean
   get("mean")
   ```

```
evalq(mean)
match.fun("mean")
```

3. By default, base R data import functions, like `read.csv()`, will automatically convert non-syntactic names to syntactic names. Why might this be problematic? What option allows you to suppress this behaviour?

4. What rules does `make.names()` use to convert non-syntactic names into syntactic names?

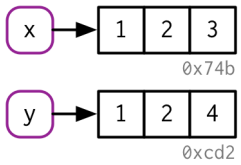
5. I slightly simplified the rules that govern syntactic names. Why is `.123e1` not a syntactic name? Read `?make.names` for the full details.

2.3 Copy-on-modify

Consider the following code, which binds `x` and `y` to the same underlying value, then modifies[2] `y`.

```
x <- c(1, 2, 3)
y <- x

y[[3]] <- 4
x
#> [1] 1 2 3
```

Modifying `y` clearly doesn't modify `x`, so what happened to the shared binding? While the value associated with `y` changes, the original object does not. Instead, R creates a new object, `0xcd2`, a copy of `0x74b` with one value changed, then rebinds `y` to that object.



[2] You may be surprised to see `[[` used with a numeric vector. We'll come back to this in Section **??**, but in brief, I think you should use `[[` whenever you are getting or setting a single element.

This behaviour is called **copy-on-modify**, and understanding it makes your intuition for the performance of R code radically better. A related way to describe this phenomenon is to say that R objects are **immutable**, or unchangeable. However, I'll generally avoid that term because there are a couple of important exceptions to copy-on-modify that you'll learn about in Section **??**.

2.3.1 `tracemem()`

You can see when an object gets copied with the help of `base::tracemem()`. You call it with an object and it returns the current address of the object:

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f80c0e0ffc8>
```

Whenever that object is copied in the future, `tracemem()` will print out a message telling you which object was copied, what the new address is, and the sequence of calls that lead to the copy:

```
y <- x
y[[3]] <- 4L
#> tracemem[0x7f80c0e0ffc8 -> 0x7f80c4427f40]:
```

Note that if you modify `y` again, it doesn't get copied. That's because the new object now only has a single name binding to it, so R can apply a modify-in-place optimisation. We'll come back to that shortly.

```
y[[3]] <- 5L

untracemem(y)
```

`untracemem()` is the opposite of `tracemem()`; it turns tracing off.

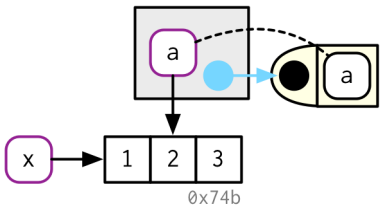
2.3.2 Function calls

The same rules for copying also apply to function calls. Take this code:

```
f <- function(a) {
  a
}
```

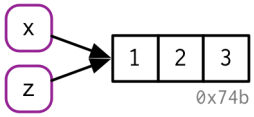
```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7ffb271c7548>

z <- f(x)
# there's no copy here!

untracemem(x)
```

While `f()` is running, `a` inside the function will point to the same value as `x` does outside of it:



(You'll learn more about the conventions used in this diagram in Execution environments.)

And once complete, `x` and `z` will point to the same object. `0x74b` never gets copied because it never gets modified. If `f()` did modify `x`, R would create a new copy, and then `z` would bind that object.

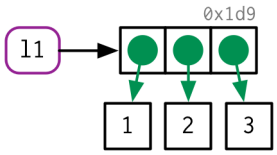


2.3.3 Lists

It's not just names (i.e. variables) that point to values; the elements of lists do too. Take this list, which superficially is very similar to the vector above:

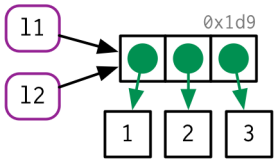
```
l1 <- list(1, 2, 3)
```

The internal representation of the list is actually quite different to that of a vector. A list is really a vector of references:

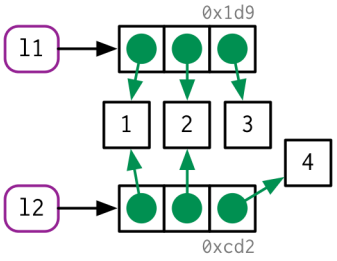


This is particularly important when we modify a list:

```
l2 <- l1
```




```
l2[[3]] <- 4
```



Like vectors, lists are copied-on-modify; the original list is left unchanged, and R creates a modified copy. This is a **shallow** copy: the list object and its bindings are copied, but the values pointed to by the bindings are not. The oppposite of a shallow copy is a deep copy, where the contents of every reference are also copied. Prior to R 3.1.0, copies were always deep copies, .

You can use `lobstr::ref()` to see values that are shared across lists. `ref()` prints the memory address of each object, along with a local id so that you can easily cross-reference shared components.

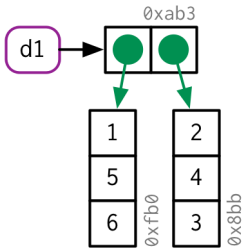
```
ref(l1, l2)
#>   [1:0x7ffb297678a8] <list>
#>   [2:0x7ffb2962d5a8] <dbl>
#>   [3:0x7ffb2962d570] <dbl>
#>   [4:0x7ffb2962d538] <dbl>
#>
```

```
#>   [5:0x7ffb2b56c6d8] <list>
#>   [2:0x7ffb2962d5a8]
#>   [3:0x7ffb2962d570]
#>   [6:0x7ffb27709550] <dbl>
```

2.3.4 Data frames

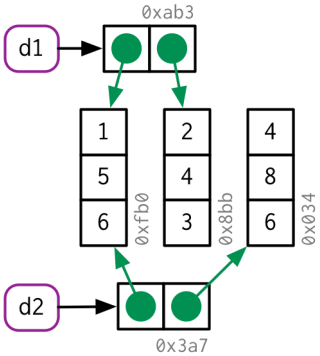
Data frames are lists of vectors, so copy-on-modify has important consequences when you modify a data frame. Take this data frame as an example:

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```



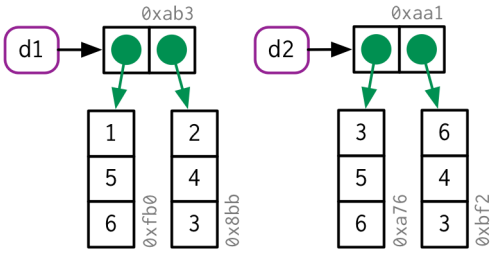
If you modify a column, only that column needs to be modified; the others can continue to point to the same place:

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```



However, if you modify a row, there is no way to share data with the previous version of the data frame, and every column must be copied-and-modified.

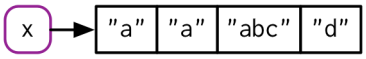
```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```



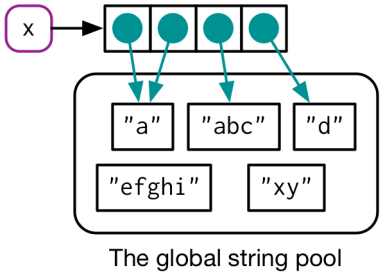
2.3.5 Character vectors

The final place that R uses references is in character vectors. I usually draw character vectors like this:

```
x <- c("a", "a", "abc", "d")
```



But this is a polite fiction, because R has a **global string pool**. Each element of a character vector is actually a pointer to a unique string in that pool:



You can request that `ref()` show these references:

```
ref(x, character = TRUE)
#>   [1:0x7ffb283dc4c8] <chr>
#>   [2:0x7ffb268e4398] <string: "a">
#>   [2:0x7ffb268e4398]
```

```
#>   [3:0x7ffb2937c118] <string: "abc">
#>   [4:0x7ffb26dda748] <string: "d">
```

This has a profound impact on the amount of memory a character vector takes but, but is otherwise not generally important, so elsewhere in the book I'll draw character vectors as if the strings live inside the vector.

2.3.6 Exercises

1. Why is `tracemem(1:10)` not useful?
2. Explain why `tracemem()` shows two copies when you run this code. Hint: carefully look at the difference between this code and the code shown earlier in the section.

```
x <- c(1L, 2L, 3L)
tracemem(x)

x[[3]] <- 4
```

3. Sketch out the relationship between the following objects:

```
a <- 1:10
b <- list(a, a)
c <- list(b, a, 1:10)
```

4. What happens when you run this code?

```
x <- list(1:10)
x[[2]] <- x
```

   Draw a picture.

2.4 Object size

You can find out how much space an object occupies in memory with `lobstr::obj_size()`[3]:

```
obj_size(letters)
#> 1,792 B
obj_size(ggplot2::diamonds)
#> 3,457,104 B
```

Since the elements of lists are references to values, the size of a list might be much smaller than you expect:

```
x <- runif(1e6)
obj_size(x)
#> 8,000,048 B

y <- list(x, x, x)
obj_size(y)
#> 8,000,128 B
```

`y` is only 72 bytes[4] bigger than `x`. That's the size of an empty list with three elements:

```
obj_size(list(NULL, NULL, NULL))
#> 80 B
```

Similarly, the global string pool means that character vectors take up less memory than you might expect: repeating a string 1000 times does not make it take up 1000 times as much memory.

```
banana <- "bananas bananas bananas"
obj_size(banana)
#> 272 B
obj_size(rep(banana, 100))
#> 1,064 B
```

[3]Beware of the base `utils::object.size()` function. It does not correctly account for shared references and will return sizes that are too large.

[4]If you're running 32-bit R you'll see slightly different sizes.

References also make it challenging to think about the size of individual objects. `obj_size(x) + obj_size(y)` will only equal `obj_size(x, y)` if there are no shared values. Here, the combined size of `x` and `y` is the same as the size of `y`:

```
obj_size(x, y)
#> 8,000,128 B
```

2.4.1 Exercises

1. In the following example, why are `object.size(y)` and `obj_size(y)` so radically different? Consult the documentation of `object.size()`.

```
y <- rep(list(runif(1e4)), 100)

object.size(y)
#> 8005648 bytes
obj_size(y)
#> 80,896 B
```

2. Take the following list. Why is its size somewhat misleading?

```
x <- list(mean, sd, var)
obj_size(x)
#> 17,664 B
```

3. Predict the output of the following code:

```
x <- runif(1e6)
obj_size(x)

y <- list(x, x)
obj_size(y)
obj_size(x, y)

y[[1]][[1]] <- 10
obj_size(y)
obj_size(x, y)
```

```
y[[2]][[1]] <- 10
obj_size(y)
obj_size(x, y)
```

2.5 Modify-in-place

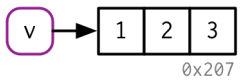
As we've seen above, modifying an R object will usually create a copy. There are two exceptions that we'll explore below:

- Objects with a single binding get a special performance optimisation.
- Environments are a special type of object that is always modified in place.

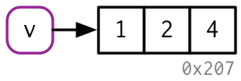
2.5.1 Objects with a single binding

If an object only has a single name that binds it, R will modify it in place:

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




```
v[[3]] <- 4
```



(Carefully note the object ids here: `v` continues to bind to the same object, `0x207`.)

It's challenging to predict exactly when R applies this optimisation because of two complications:

- When it comes to bindings, R can currently[5] only count 0, 1, and many.

[5] By the time you read this, that may have changed, as plans are afoot to improve reference counting: `https://developer.r-project.org/Refcnt.html`

That means if an object has two bindings, and one goes away, the reference count does not go back to 1 (because one less than many is still many).

- Whenever you call any regular function, it will make a reference to the object. The only exception are specially written C functions. These occur mostly in the base package.

Together, this makes it hard to predict whether or not a copy will occur. Instead, it's better to determine it empirically with `tracemem()`. Let's explore the subtleties with a case study using for loops. For loops have a reputation for being slow in R, but often that slowness is because every iteration of the loop is creating a copy.

Consider the following code. It subtracts the median from each column of a large data frame:

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians)) {
  x[[i]] <- x[[i]] - medians[[i]]
}
```

This loop is surprisingly slow because every iteration of the loop copies the data frame, as revealed by using `tracemem()`:

```
cat(tracemem(x), "\n")
#> <0x7f80c429e020>

for (i in 1:5) {
  x[[i]] <- x[[i]] - medians[[i]]
}
#> tracemem[0x7f80c429e020 -> 0x7f80c0c144d8]:
#> tracemem[0x7f80c0c144d8 -> 0x7f80c0c14540]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14540 -> 0x7f80c0c145a8]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c145a8 -> 0x7f80c0c14610]:
#> tracemem[0x7f80c0c14610 -> 0x7f80c0c14678]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14678 -> 0x7f80c0c146e0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c146e0 -> 0x7f80c0c14748]:
#> tracemem[0x7f80c0c14748 -> 0x7f80c0c147b0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c147b0 -> 0x7f80c0c14818]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14818 -> 0x7f80c0c14880]:
#> tracemem[0x7f80c0c14880 -> 0x7f80c0c148e8]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c148e8 -> 0x7f80c0c14950]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14950 -> 0x7f80c0c149b8]:
```

```
#> tracemem[0x7f80c0c149b8 -> 0x7f80c0c14a20]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14a20 -> 0x7f80c0c14a88]: [[<-.data.frame [[<-

untracemem(x)
```

In fact, each iteration copies the data frame not once, not twice, but three times! Two copies are made by `[[.data.frame`, and a further copy[6] it made because `[[.data.frame` is a regular function and hence increments the reference count of `x`.

We can reduce the number of copies by using a list instead of a data frame. Modifying a list uses internal C code, so the refs are not incremented and only a single copy is made:

```
y <- as.list(x)
cat(tracemem(y), "\n")
#> <0x7f80c5c3de20>

for (i in 1:5) {
  y[[i]] <- y[[i]] - medians[[i]]
}
#> tracemem[0x7f80c5c3de20 -> 0x7f80c48de210]:
```

While it's not hard to determine when copies are made, it is hard to prevent them. If you find yourself resorting to exotic tricks to avoid copies, it may be time to rewrite your function in C++, as described in Chapter **??**.

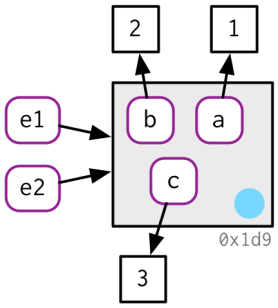
2.5.2 Environments

You'll learn more about environments in Chapter **??**, but it's important to mention them here because they behave differently to other objects: environments are always modified in place. This property is sometimes described as **reference semantics** because when you modify an environment all existing bindings to the environment continue to have the same reference.

Take this environment, which we bind to `e1` and `e2`:

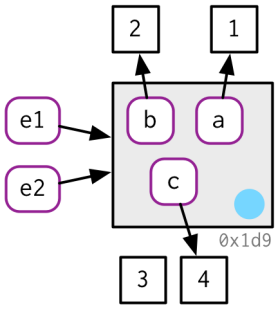
```
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
```

[6] Note that these copies are shallow, and only copy the reference to each individual column, not the contents. This means the performance isn't terrible, but it's obviously not as good as it could be.



If we change a binding, the environment is modified in place:

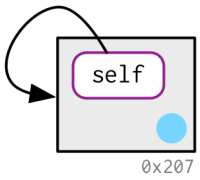
```
e1$c <- 4
e2$c
#> [1] 4
```



This basic idea can be used to create functions that "remember" their previous state. See Section **??** for more details.

One consequence of this is that environments can contain themselves:

```
e <- rlang::env()
e$self <- e

ref(e)
#>   [1:0x7ffb2c263a10] <env>
#>   self = [1:0x7ffb2c263a10]
```



This is a unique property of environments!

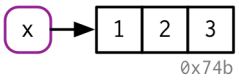
2.5.3 Exercises

1. Wrap the two methods for subtracting medians into two functions, then use the bench (**?**) package to carefully compare their speeds. How does performance change as the number of columns increase?
2. What happens if you attempt to use `tracemem()` on an environment?

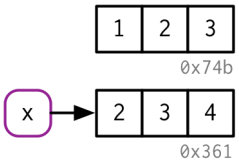
2.6 Unbinding and the garbage collector

Consider this code:

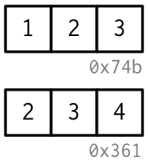
```
x <- 1:3
```




```
x <- 2:4
```




```
rm(x)
```



We create two objects, but by the end of code neither object is bound to a name. How do these objects get deleted? That's the job of the **garbage collector**, or GC, for short. The GC creates more memory by deleting R objects that are no longer used, and if needed, requesting more memory from the operating system.

R uses a **tracing** GC. That means it traces every object reachable from the global[7] environment, and all the objects reachable from those objects (i.e. the references in lists and environments are searched recursively). The garbage collector does not use the reference count used for the modify-in-place optimisation described above. The two ideas are closely related but the internal data structures have been optimised for different use cases.

The garbage collector (GC) is run automatically whenever R needs more memory to create a new object. From the outside, it's basically impossible to predict when the GC will run, and indeed, you shouldn't try. Instead, if you want to find out when the GC runs, call `gcinfo(TRUE)`: the the GC will print a message to the console every time it runs.

You can force the garbage collector to run by calling `gc()`. Despite what you might have read elsewhere, there's never any *need* to call `gc()` yourself. You may *want* to call `gc()` to ask R to return memory to your operating system, or for its side-effect of telling you how much memory is currently being used:

```
gc()
#>           used (Mb) gc trigger  (Mb) limit (Mb) max used  (Mb)
#> Ncells  674683 36.1    1236521  66.1         NA  1236521  66.1
#> Vcells 3678440 28.1   17075913 130.3      16384 17073996 130.3
```

`lobstr::mem_used()` is a wrapper around `gc()` that just prints the total number of bytes used:

```
mem_used()
#> 67,208,856 B
```

This number won't agree with the amount of memory reported by your operating system for three reasons:

[7] And every environment on the current call stack.

1. It only includes objects created by R, not the R interpreter itself.
2. Both R and the operating system are lazy: they won't reclaim memory until it's actually needed. R might be holding on to memory because the OS hasn't yet asked for it back.
3. R counts the memory occupied by objects but there may be gaps due to deleted objects. This problem is known as memory fragmentation.

2.7 Answers

1. You must surround non-syntactic names in `. The variables `1`, `2`, and `3` have non-syntactic names, so must always be quoted with backticks.

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)

df$`3` <- df$`1` + df$`2`
```

2. It occupies about 4 MB.

```
x <- runif(1e6)
y <- list(x, x, x)
obj_size(y)
#> 8,000,128 B
```

3. `a` is copied when `b` is modified, `b[[1]] <- 10`.

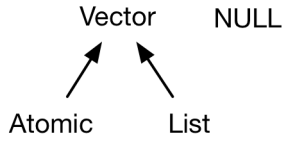
3

Vectors

3.1 Introduction

This chapter discusses the most important family of data types in base R: the vector types[1]. You've probably used many (if not all) of the vectors before, but you may not have thought deeply about how they are interrelated. In this chapter, I won't cover individual vectors types in too much 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.

Vectors come in two flavours: atomic vectors and lists[2]. 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. Closely related to vectors is `NULL`; `NULL` is not a vector, but often serves the role of a generic 0-length vector. Throughout this chapter we'll expand on this diagram:



Every vector can also have **attributes**, which you can think of as a named list containing arbitrary metadata. Two attributes are particularly important because they create important vector variants. The **dim**ension attribute turns vectors into matrices and arrays. The **class** attribute powers the S3 object system. You'll learn how to use S3 in Chapter **??**, but here, you'll learn about a handful of the most important S3 vectors: factors, date/times, data frames, and tibbles. Matrices and data frames are not necessarily what you think of

[1] Collectively, all other data types are known as the "node" data types, and includes things like functions and environments. This is a highly technical term used in only a few places. The place where you're most likely to encounter it is the output of `gc()`: the "N" in `Ncells` stands for nodes, and the "V" in `Vcells` stands for vectors.

[2] A few places in R's documentation call lists generic vectors to emphasise their difference from atomic vectors.

a vectors, so you'll learn why these 2d structures are considered to be vectors in R.

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 answers.

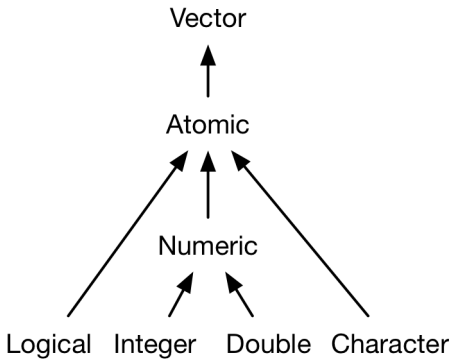
1. What are the four common types of atomic vectors? What are the two rare types?
2. What are attributes? How do you get them and set them?
3. How is a list different from an atomic vector? How is a matrix different from a data frame?
4. Can you have a list that is a matrix? Can a data frame have a column that is a matrix?
5. How do tibbles behave differently from data frames?

Outline

3.2 Atomic vectors

There are four common types of atomic vectors: logical, integer, double, and character. Collectively integer and double vectors are known as numeric vectors[3]. There are two rare types that I won't discuss further: complex and raw. Complex numbers are rarely needed for statistics, and raw vectors are a special type only needed when handling binary data.

[3]This is a slight simplification as R does not use "numeric" consistently, which we'll come back to in Section **??**.



3.2.1 Scalars

Each of the four primary atomic vectors has special syntax to create an individual value, aka a **scalar**[4], and its own missing value.:

- Strings are surrounded by `"` (`"hi"`) or `'` (`'bye'`). The string missing value is `NA_character_`. Special characters are escaped with `\\`; see `?Quotes` for full details.
- Doubles can be specified in decimal (`0.1234`), scientific (`1.23e4`), or hexadecimal (`0xcafe`) forms. There are three special values unique to doubles: `Inf`, `-Inf`, and `NaN`. The double misssing value is `NA_real_`.
- Integers are written similarly to doubles but must be followed by `L`[5] (`1234L`, `1e4L`, or `0xcafeL`), and can not include decimals. The integer missing value is `NA_integer_`.
- Logicals can be spelled out (`TRUE` or `FALSE`), or abbreviated (`T` or `F`). The logical missing value is `NA`.

3.2.2 Making longer vectors with `c()`

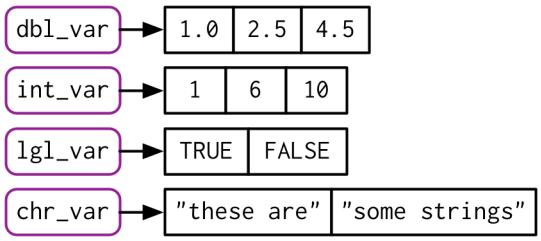
To greater longer vectors from shorter vectors, use `c()`:

[4]Technically, the R language does not possess scalars, and everything that looks like a scalar is actually a vector of length one. This however, is mainly a theoretical distinction, and blurring the distinction between scalar and length-1 vector is unlikely to harm your code.

[5]`L` is not intuitive, and you might wonder where it comes from. At the time `L` was added to R, R's integer type was equivalent to a long integer in C, and C code could use a suffix of `l` or `L` to force a number to be a long integer. It was decided that `l` was too visually similar to `i` (used for complex numbers in R), leaving `L`.

```
dbl_var <- c(1, 2.5, 4.5)
int_var <- c(1L, 6L, 10L)
lgl_var <- c(TRUE, FALSE)
chr_var <- c("these are", "some strings")
```

In diagrams, I'll depict vectors as connected rectangles, so the above code could be drawn as follows:



You can determine the type of a vector with `typeof()` and its length with `length()`.

```
typeof(dbl_var)
#> [1] "double"
typeof(int_var)
#> [1] "integer"
typeof(lgl_var)
#> [1] "logical"
typeof(chr_var)
#> [1] "character"
```

3.2.3 Testing and coercing

Generally, you can **test** if a vector is of a given type with an `is.` function, but they need to be used with care. `is.character()`, `is.double()`, `is.integer()`, and `is.logical()` do what you might expect: they test if a vector is a character, double, integer, or logical. Beware `is.vector()`, `is.atomic()`, and `is.numeric()`: they don't test if you have a vector, atomic vector, or numeric vector! We'll come back to what they actually do in Section **??**.

The type is a propety of the entire atomic vector, so all elements of an atomic must be the same type. When you attempt to combine different types they will be **coerced** to the most flexible one (character >> double >> integer >> logical). For example, combining a character and an integer yields a character:

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

Coercion often happens automatically. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to numeric. This particularly useful for logical vectors because `TRUE` becomes 1 and `FALSE` becomes 0.

```
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
```

Vectorised logical operations (`&`, `|`, `any`, etc) will coerce to a logical, but since this might lose information, it's always accompanied a warning.

Generally, you can deliberately coerce by using an `as.` function, like `as.character()`, `as.double()`, `as.integer()`, or `as.logical()`. Failed coercions from strings generate a warning and a missing value:

```
as.integer(c("1", "1.5", "a"))
#> Warning: NAs introduced by coercion
#> [1]  1  1 NA
```

3.2.4 Exercises

1. How do you create scalars of type raw and complex? (See `?raw` and `?complex`)
2. Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

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

3. Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?

4. Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

3.3 Attributes

You might have noticed that the set of atomic vectors does not include a number of important data structures like matrices and arrays, factors and date/times. These types are built on top of atomic vectors by adding attributes. In this section, you'll learn the basics of attributes, and how the dim attribute makes matrices and arrays. In the next section you'll learn how the class attribute is used to create S3 vectors, including factors, dates, and date-times.

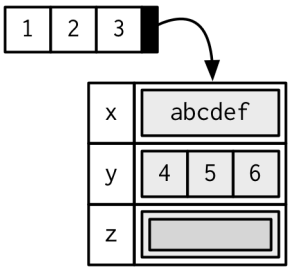
3.3.1 Getting and setting

You can think of attributes as a named list[6] used to attach metadata to an object. Individual attributes can be retrieved and modified with `attr()`, or retrieved en masse with `attributes()`, and set en masse with `structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
attr(a, "x")
#> [1] "abcdef"

attr(a, "y") <- 4:6
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6

# Or equivalently
a <- structure(
  1:3,
  x = "abcdef",
```

[6]The reality is a little more complicated: attributes are actually stored in pairlists. Pairlists are functionally indistinguisable from lists, but are profoundly different under the hood, and you'll learn more about them in Section **??**.

```
  y = 4:6
)
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6
```



Attributes should generally be thought of as ephemeral. For example, most attributes are lost by most operations:

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

There are only two attributes that are routinely preserved:

- **names**, a character vector giving each element a name.
- **dims**, short for dimensions, an integer vector, used to turn vectors into matrices and arrays.

To preserve additional attributes, you'll need to create your own S3 class, the topic of Chapter **??**.

3.3.2 Names

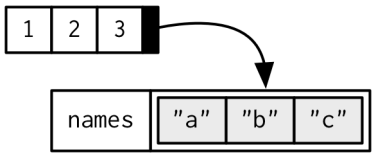
You can name a vector in three ways:

```
# When creating it:
x <- c(a = 1, b = 2, c = 3)

# By assigning names() to an existing vector:
x <- 1:3
```

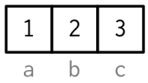
```
names(x) <- c("a", "b", "c")

# Inline, with setNames():
x <- setNames(1:3, c("a", "b", "c"))
```

Avoid using `attr(x, "names")` as it more typing and less readable than `names(x)`. You can remove names from a vector by using `unname(x)` or `names(x) <- NULL`.

To be technically correct, when drawing the named vector `x`, I should draw it like so:



However, names are so special and so important, that unless I'm trying specifically to draw attention to the attributes data structure, I'll use them to label the vector directly:



To be maximally useful for character subsetting (e.g. Section **??**) names should be unique, and non-missing, but this is not enforced by R. Depending on how the names are set, missing names may be either `""` or `NA_character_`. If all names are missing, `names()` will return `NULL`.

3.3.3 Dimensions

Adding a `dim` attribute to a vector allows it to behave like a 2-dimensional **matrix** or multi-dimensional **array**. Matrices and arrays are primarily a mathematical/statistical tool, not a programming tool, so will be used infrequently in this book, and only covered briefly. Their most important feature of is multidimensional subsetting, which is covered in Section **??**.

You can create matrices and arrays with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments specify row and column sizes
a <- matrix(1:6, nrow = 2, ncol = 3)
```

```
a
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6

# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))
b
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12

# 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
```

Many of the functions for working with vectors have generalisations for matrices and arrays:

	Vector	Matrix	Array
`names()`	`rownames()`, `colnames()`	`dimnames()`
`length()`	`nrow()`, `ncol()`	`dim()`
`c()`	`rbind()`, `cbind()`	`abind::abind()`
—	`t()`	`aperm()`
`is.null(dim(x))`	`is.matrix()`	`is.array()`

A vector without `dim` attribute set is often thought of as 1-dimensional, but actually has a `NULL` dimensions. You also 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
```

3.3.4 Exercises

1. How is `setNames()` implemented? How is `unname()` implemented? Read the source code.
2. What does `dim()` return when applied to a 1d vector? When might you use `NROW()` or `NCOL()`?
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))
   ```

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

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

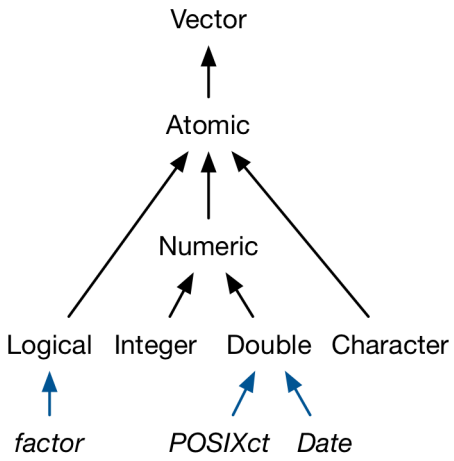
   But when you print that object you don't see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.)

3.4 S3 atomic vectors

One of the most important attributes is `class`, which defines the S3 object system. Having a class attribute makes an object an **S3 object**, which means that it will behave differently when passed to a **generic** function. Every S3 object is built on top of a base type, and often stores additional information in other attributes. You'll learn the details of the S3 object system, and how to create your own S3 classes, in Chapter **??**.

In this section, we'll discuss three important S3 vectors used in base R:

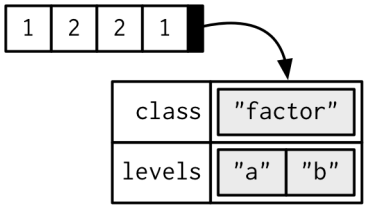
- Categorical data, where values can only come from a fixed set of levels, are recorded in **factor** vectors.
- Dates (with day resolution) are recorded are **Date** vectors.
- Date-times (with second or sub-second) resolution are stored in **POSIXct** vectors.



3.4.1 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 with 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

typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "a" "b"
#>
#> $class
#> [1] "factor"
```



Factors are useful when you know the set of possible values, even if you don't see them all in a given dataset. Compared to a character vector, this means that tabulating a factor can yield counts of 0:

```
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
```

A minor variation or factors is **ordered** factors, which generally behave similarly, but declare that the order of the levels is meaningful (a fact which is used automatically in some models and visualisations).

```
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
#> [1] b b a c
#> Levels: c < b < a
```

With base R[7] you tend to encouter factors very frequently, because many base R functions (like `read.csv()` and `data.frame()`) 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: the levels are a property of the experimental design, not the data. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data. To learn about the historical context of this behaviour, I recommend *stringsAsFactors: An unauthorized biography* (`http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/`) by Roger Peng, and *stringsAsFactors = <sigh>* (`http://notstatschat.tumblr.com/post/124987394001/stringsasfactors-sigh`) by Thomas Lumley.

While factors look like (and often behave like) character vectors, they are built on top of integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings automatically, 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.

3.4.2 Dates

Date vectors are built on top of double vectors. They have class "Date" and no other attributes:

```
today <- Sys.Date()

typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date"
```

The value of the double (which can be seen by stripping the class), represents the number of days since 1970-01-01:

[7] The tidyverse never automatically coerce characters to factor, and provides the forcats (?) package specifically for working with factors.

```
date <- as.Date("1970-02-01")
unclass(date)
#> [1] 31
```

3.4.3 Date-times

Base R[^tidyverse-datetimes] provides two ways of storing date-time information, POSIXct, and POSIXlt. These are admittedly odd names: "POSIX" is short for Portable Operating System Interface which is a family of cross-platform standards. "ct" standards for calendar time (the `time_t` type in C), and "lt" for local time (the `struct tm` type in C). Here we'll focus on `POSIXct`, because it's the simplest, is built on top of an atomic vector, and is most appropriate for use in data frames. POSIXct vectors are built on top of double vectors, where the value represents the number of days since 1970-01-01.

```
now_ct <- as.POSIXct("2018-08-01 22:00", tz = "UTC")
now_ct
#> [1] "2018-08-01 22:00:00 UTC"

typeof(now_ct)
#> [1] "double"
attributes(now_ct)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
```

The `tzone` attribute controls how the date-time is formatted, not the instant of time represented by the vector. Note that the time is not printed if it is midnight.

```
structure(now_ct, tzone = "Asia/Tokyo")
#> [1] "2018-08-02 07:00:00 JST"
structure(now_ct, tzone = "America/New_York")
#> [1] "2018-08-01 18:00:00 EDT"
structure(now_ct, tzone = "Australia/Lord_Howe")
#> [1] "2018-08-02 08:30:00 +1030"
structure(now_ct, tzone = "Europe/Paris")
#> [1] "2018-08-02 CEST"
```

[^tidyverse-datetimes] The tidyverse provides the lubridate (**?**) package for

working with date-times. It provides a number of convenient helpers all which work with the base POSIXct type.

3.4.4 Exercises

1. What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensionality change as you tabulate more variables?
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))
```

3.5 Lists

Lists are a step up in complexity from atomic vectors because an element of a list can be any type (not just vectors). An element of a list can even be another list!

3.5.1 Creating

Construct lists with `list()`:

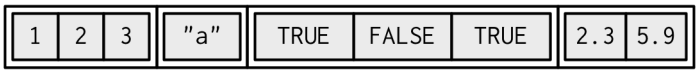
```
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9)
)
```

```
typeof(l1)
#> [1] "list"

str(l1)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

As described in Section **??**, the elements of a list are references. Creating a list does not copy the components in, so the total size of a list might be smaller than you expect.

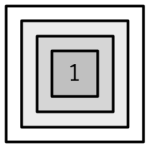
```
lobstr::obj_size(mtcars)
#> 7,792 B

l2 <- list(mtcars, mtcars, mtcars, mtcars)
lobstr::obj_size(l2)
#> 7,872 B
```

Lists can contain complex objects so it's not possible to pick one visual style that works for every list. Generally I'll draw lists like vectors, using colour to remind you of the hierarchy.



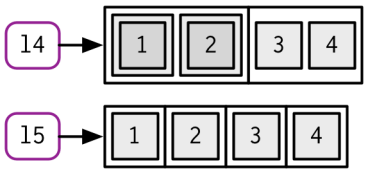
Lists are sometimes called **recursive** vectors, because a list can contain other lists. This makes them fundamentally different from atomic vectors.

```
l3 <- list(list(list(1)))
str(l3)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : num 1
```



`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()`:

```
l4 <- list(list(1, 2), c(3, 4))
l5 <- c(list(1, 2), c(3, 4))
str(l4)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(l5)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```



3.5.2 Testing and coercing

The `typeof()` a list is `list`. You can test for a list with `is.list()`.

And coerce to a list with `as.list()`.

```
list(1:3)
#> [[1]]
#> [1] 1 2 3
as.list(1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
```

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

You can turn a list into an atomic vector with `unlist()`. The rules for the resulting type are complex, not well documented, and not always equivalent to `c()`.

3.5.3 Matrices and arrays

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

l[[1, 1]]
#> [1] 1 2 3
```

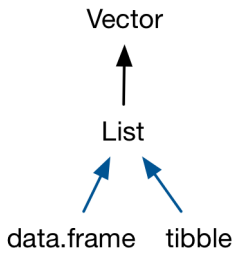
These are relatively esoteric data structures, but can be useful if you want to arrange objects into a grid-like structure. For example, if you're running models on a spatio-temporal grid, it might be natural to preserve the grid structure by storing the models in a 3d array.

3.5.4 Exercises

1. List all the ways that a list differs from an atomic vector?
2. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?
3. Compare and contrast `c()` and `unlist()` when combining a date and date-time into a single vector.

3.6 Data frames and tibbles {tibble}

There are two important S3 vectors that are built on top of lists: data frames and tibbles.



A data frame is the most common way of storing data in R, and is crucial for effective data analysis. A data frames is a named list of equal-length vectors. It has attributes providing the (column) `names`, `row.names`[8], and a class of "data.frame":

```
df1 <- data.frame(x = 1:2, y = 2:1)
typeof(df1)
#> [1] "list"

attributes(df1)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2
```

Because each element of the list has the same length, data frames have a rectangular structure, and hence shares properties of both the matrix and the list:

[8]Row names are one of the most suprisingly complex data structures in R, because they've been persistent performance issue over many years. The most straightforward representations are character or integer vectors, with one element for each row. There's also a compact representation for "automatic" row names (consecutive integers), created by `.set_row_names()`. R 3.5 has a special way of deferring integer to character conversions specifically to speed up `lm()`; see `https://svn.r-project.org/R/branches/ALTREP/ALTREP.html#deferred_string_conversions` for details.

- A data frame has 1d `names()`, and 2d `colnames()` and `rownames()`[9]. The `names()` and `colnames()` are identical.
- A data frame has 1d `length()`, and 2d `ncol()` and `nrow()`. The `length()` is the number of columns.

Data frames are one of the biggest and most important ideas in R, and one of the things that makes R different from other programming languages. However, in the over 20 years since their creation, the ways people use R have changed, and some of the design decisions that made sense at the time data frames were created now cause frustration.

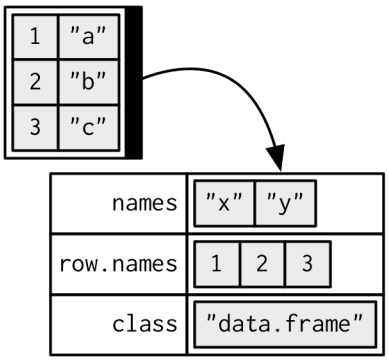
This frustration lead to the creation of the tibble (**?**), a modern reimagining of the data frame. Tibbles are designed to be (as much as possible) drop-in replacements for data frames, while still fixing the greatest frustrations. A concise, and fun, way to summarise the main differences is that tibbles are lazy and surly: they tend to do less and complain more. You'll see what that means as you work through this section.

Tibbles are provided by the tibble package and share the the same structure as a data frame. The only difference is that the class vector is longer, and includes `tbl_df`. This allows tibbles to behave differently in the key ways which we'll discuss below.

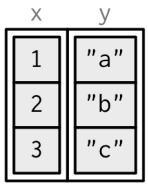
```
library(tibble)

df2 <- tibble(x = 1:2, y = 2:1)
typeof(df2)
#> [1] "list"

attributes(df2)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2
#>
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
```

When drawing data frames and tibbles, rather than focussing on the implementation details, i.e. the attributes:

[9] Technically, you are encouraged to use `row.names()`, not `rownames()` with data frames, but this distinction is rarely important.



I'll draw them in the same way as a named list, but arranged to emphasised their columnar structure.



3.6.1 Creating

You create a data frame by supplying name-vector pairs to `data.frame()`:

```
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 the default conversion of strings to factors. Use `stringsAsFactors = FALSE` to suppress it and keep character vectors as character vectors:

```
df1 <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE
```

```
)
str(df1)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Creating a tibble is similar, but tibbles never coerce their input (this is one feature that makes them lazy):

```
df2 <- tibble(
  x = 1:3,
  y = c("a", "b", "c")
)
str(df2)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Additionally, while data frames automatically transform non-syntactic names (unless `check.names = FALSE`); tibbles do not (although they do print non-syntactic names surrounded by `).

```
names(data.frame(`1` = 1))
#> [1] "X1"

names(tibble(`1` = 1))
#> [1] "1"
```

While every element of a data frame (or tibble) must have the same length, both `data.frame()` and `tibble()` can recycle shorter inputs. Data frames automatically recycle columns that are an integer multiple of the longest column; tibbles only ever recycle vectors of length 1.

```
data.frame(x = 1:4, y = 1:2)
#>   x y
#> 1 1 1
#> 2 2 2
#> 3 3 1
#> 4 4 2
data.frame(x = 1:4, y = 1:3)
#> Error in data.frame(x = 1:4, y = 1:3):
#>   arguments imply differing number of rows: 4, 3
```

```
tibble(x = 1:4, y = 1)
#> # A tibble: 4 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     1
#> 4     4     1
tibble(x = 1:4, y = 1:2)
#> Error: Column `y` must be length 1 or 4, not 2
```

There is one final difference: `tibbles()` allow you to refer to newly created variables:

```
tibble(
  x = 1:3,
  y = x * 2
)
#> # A tibble: 3 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
```

3.6.2 Row names

Data frames allow you to label each row with a "name", a character vector containing only unique values:

```
df3 <- data.frame(
  age = c(35, 27, 18),
  hair = c("blond", "brown", "black"),
  row.names = c("Bob", "Susan", "Sam")
)
df3
#>       age  hair
#> Bob    35 blond
#> Susan  27 brown
#> Sam    18 black
```

You can get and set row names with `rownames()`, and you can use them to subset rows:

```
rownames(df3)
#> [1] "Bob"   "Susan" "Sam"

df3["Bob", ]
#>     age  hair
#> Bob  35 blond
```

Row names arise naturally if you think of data frames as 2d structures like matrices: the columns (variables) have names so the rows (observations) should too. Most matrices are numeric, so having a place to store character labels is important. But this analogy to matrices is misleading because matrices possess an important property that data frames do not: they are transposable. In matrices the rows and columns are interchangeable, and transposing a matrix gives you another matrix (and transposing again gives you back the original matrix). With data frames, however, the rows and columns are not interchangeable, and the transpose of a data frame is not a data frame.

There are three reasons that row names are them suboptimal:

- Metadata is data, so storing it in a different way to the rest of the data is fundamentally a bad idea. It also means that you need to learn a new set of tools to work with row names; you can't use what you already know about manipulating columns.
- Row names are poor abstraction for labelling rows because they only work when a row can be identified by a single string. This fails in many cases, for example when you want to identify a row by a non-character vector (e.g. a time point), or with multiple vectors (e.g. position, encoded by latitidue and longitude).
- Row names must be unique, so any replication of rows (e.g. from bootstrapping) will create new row names. If you want to match rows from before and after the transformation you'll need to perform complicated string surgery.

```
df3[c(1, 1, 1), ]
#>       age  hair
#> Bob    35 blond
#> Bob.1  35 blond
#> Bob.2  35 blond
```

For these reasons, tibbles do not support row names. Instead tibble package provides tools to easily convert row names into a regular column with either `rownames_to_column()`, or the `rownames` argument to `as_tibble()`:

```
as_tibble(df3, rownames = "name")
#> # A tibble: 3 x 3
#>   name    age hair
#>   <chr> <dbl> <fct>
#> 1 Bob      35 blond
#> 2 Susan    27 brown
#> 3 Sam      18 black
```

3.6.3 Printing

One of the most obvious differences between tibbles and data frames is how they are printed. I assume that youu're already familiar with how data frames are printed, so here I'll highlight some of the biggest differences using an example dataset included in the dplyr package:

```
dplyr::starwars
#> # A tibble: 87 x 13
#>    name  height  mass hair_color skin_color eye_color birth_year
#>    <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl>
#>  1 Luke~    172    77 blond      fair       blue            19
#>  2 C-3PO    167    75 <NA>       gold       yellow         112
#>  3 R2-D2     96    32 <NA>       white, bl~ red             33
#>  4 Dart~    202   136 none       white      yellow          41.9
#>  5 Leia~    150    49 brown      light      brown           19
#>  6 Owen~    178   120 brown, gr~ light      blue            52
#>  7 Beru~    165    75 brown      light      blue            47
#>  8 R5-D4     97    32 <NA>       white, red red             NA
#>  9 Bigg~    183    84 black      light      brown           24
#> 10 Obi-~    182    77 auburn, w~ fair       blue-gray       57
#> # ... with 77 more rows, and 6 more variables: gender <chr>,
#> #   homeworld <chr>, species <chr>, films <list>, vehicles <list>,
#> #   starships <list>
```

Tibbles:

- Only show the first 10 rows and all the columns that will fit on screen. Additional columns are shown at the bottom.
- Each column is labelled with its type, abbreviated to three or four letters.
- Wide columns are truncated to avoid a single long string occupying an entire row. (This is still a work in progress: it's tricky to get the tradeoff right between showing as many columns as possible and showing a single wide column fully.)

- When used in console environments that support it, colour is used judiciously to highlight important information, and de-emphasise supplemental details.

3.6.4 Subsetting

As you will learn in Chapter **??**, you can subset a data frame or a tibble like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

In my opinion, data frames have two suboptimal subsetting behaviours:

- When you subset columns with `df[, vars]`, you will get a vector if `vars` selects one variable, otherwise you'll get a data frame. This is a frequent source of bugs when using `[` in a function, unless you always remember to do `df[, vars, drop = FALSE]`.
- When you attempt to extract a single column with `df$x` and there is no column `x`, a data frame will instead select any variable that starts with `x`. If no variable starts with `x`, `df$x` will return `NULL`. This makes it easy to select the wrong variable or to select a variable that doesn't exist.

Tibbles tweak these behaviours so that `[` always returns a tibble, and `$` doesn't partial match, and warns if it can't find a variable (this is what makes tibbles surly).

```
df1 <- data.frame(xyz = "a")
df2 <- tibble(xyz = "a")

str(df1$x)
#>  Factor w/ 1 level "a": 1
str(df2$x)
#> Warning: Unknown or uninitialised column: 'x'.
#>  NULL
```

A tibble's insistence on returning a data frame from `[` can cause problems with legacy code, which often uses `df[, "col"]` to extract a single column. To fix this, use `df[["col"]]` instead; this is more expressive (since `[[` always extracts a single element) and works with both data frames and tibbles.

3.6.5 Testing and coercing

To check if an object is a data frame or tibble, use `is.data.frame()`:

```
is.data.frame(df1)
#> [1] TRUE
is.data.frame(df2)
#> [1] TRUE
```

Typically, it should not matter if you have a tibble or data frame, but if you do need to distinguish, use `is_tibble()`:

```
is_tibble(df1)
#> [1] FALSE
is_tibble(df2)
#> [1] TRUE
```

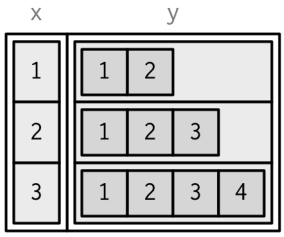
You can coerce an object to a data frame with `as.data.frame()` or to as tibble with `as_tibble()`.

3.6.6 List 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. This very useful because a list can contain any other object, which means that you can put any object in a data frame. This allows you to keep related objects together in a row, no matter how complex the individual objects are. You can see an application of this in the "Many Models" chapter of "R for Data Sicence", `http://r4ds.had.co.nz/many-models.html`.

List-columns are allowed in data frames but you have to do a little extra work, either adding the list-column after creation, or wrapping the list in `I()`.

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

data.frame(
  x = 1:3,
  y = I(list(1:2, 1:3, 1:4))
)
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```



List columns are easier to use with tibbles because you can provide them inside `tibble()`, are they are handled specially when printing:

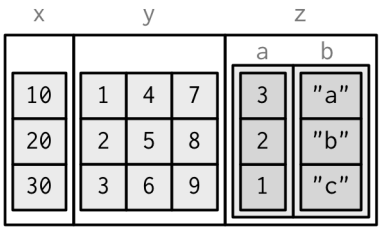
```
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4)
)
#> # A tibble: 3 x 2
#>       x y
#>   <int> <list>
#> 1     1 <int [2]>
#> 2     2 <int [3]>
#> 3     3 <int [4]>
```

3.6.7 Matrix and data frame columns

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. (This requires a slight extention to our definition of a data frame: it's not the `length()` of each column that must be equal; but the `NROW()`.) Like with list-columns, you must either add after creation, or wrap in `I()`.

```
dfm <- data.frame(
  x = 1:3 * 10
)
dfm$y <- matrix(1:9, nrow = 3)
dfm$z <- data.frame(a = 3:1, b = letters[1:3], stringsAsFactors = FALSE)

str(dfm)
#> 'data.frame':    3 obs. of  3 variables:
#>  $ x: num  10 20 30
#>  $ y: int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
#>  $ z:'data.frame':   3 obs. of  2 variables:
```

```
#>   ..$ a: int  3 2 1
#>   ..$ b: chr  "a" "b" "c"
```



Matrix and data frame columns require a little caution. Many functions that work with data frames assume that all columns are vectors, and the printed display can be confusing.

```
dfm[1, ]
#>    x y.1 y.2 y.3 z.a z.b
#> 1 10   1   4   7   3   a
```

3.6.8 Exercises

1. Can you have a data frame with 0 rows? What about 0 columns?
2. What happens if you attempt to set rownames that are not unique?
3. If `df` is a data frame, what can you say about `t(df)`, and `t(t(df))`? Perform some experiments, making sure to try different column types.
4. What does `as.matrix()` do when applied to a data frame with columns of different types? How does it differ from `data.matrix()`?

3.7 NULL

To finish up the chapter, I wanted to talk about a final important data struture that's closely related to vectors: `NULL`. `NULL` is special because it has a unique type, is always length 0, and can't have any attributes:

```
typeof(NULL)
#> [1] "NULL"

length(NULL)
#> [1] 0

x <- NULL
attr(x, "y") <- 1
#> Error in attr(x, "y") <- 1:
#>   attempt to set an attribute on NULL
```

You can test for NULLs with `is.null()`:

```
is.null(NULL)
#> [1] TRUE
```

There are two common uses of `NULL`:

- To represent an empty vector (a vector of length 0) of arbitrary type. For example, if you use `c()` but don't include any arguments, you get `NULL`, and concatenating `NULL` to a vector leaves it unchanged[10]:

  ```
  c()
  #> NULL
  ```

- To represent an absent vector. For example, `NULL` is often used as a default function argument, when the argument is optional but the default value requires some computation (see Section **??** for more on this idea). Contrast this with `NA` which is used to indicate that an *element* of a vector is absent.

If you're familiar with SQL, you know about relational `NULL` and might expect it to be the same as Rs. However, the database `NULL` is actually equivalent to `NA`.

3.8 Answers

1. The four common types of atomic vector are logical, integer, double and character. The two rarer types are complex and raw.

[10] Algebraically, this makes `NULL` the identity element under vector concatenation.

2. 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()`.

3. 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.

4. You can make "list-array" by assigning 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()))`.

5. Tibbles have an enhanced print method, never coerce strings to factors, and provide stricter subsetting methods.

4

Subsetting

4.1 Introduction

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 easy to learn but hard to master because you need to internalise a number of interrelated concepts:

- The six types of thing that you can subset with.
- The three subsetting operators, `[[`, `[`, and `$`.
- How the subsetting operators interact with vector types (e.g., atomic vectors, lists, factors, matrices, and data frames).
- The use of subsetting together 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. For large, complex objects, I also highly recommend the interactive RStudio Viewer, which you can activate with `View(my_object)`.

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 **??**.

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 **??** 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, and data frames.
- Section **??** expands your knowledge of subsetting operators to include `[[` and `$`, focussing on the important principles of simplifying vs. preserving.
- In Section **??** you'll learn the art of subassignment, combining subsetting and assignment to modify parts of an object.
- Section **??** leads you through eight important, but not obvious, applications of subsetting to solve problems that you often encounter in a data analysis.

4.2 Selecting multiple elements

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 which allows you to extract any number of elements. Section **??** will cover `[[` and `$`, used to extra a single element from a data structure.

4.2.1 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 six 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 can write an 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 silently **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 1d vectors, as you'll see shortly, 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
```

4.2.2 Lists

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

4.2.3 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(TRUE, FALSE, TRUE), 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. You'll learn how to avoid this in Section **??**.

Because matrices and arrays are just vectors with special attributes, you can subset them with a single vector, as if they were a 1d vector. Arrays in R are stored in column-major order:

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

```
select <- rbind(
  c(1, 1),
  c(3, 1),
  c(2, 4)
)
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

4.2.4 Data frames and tibbles

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, which
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
```

Subsetting a tibble with `[` always returns a tibble:

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

str(df["x"])
#> Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
str(df[, "x"])
#> Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
```

4.2.5 Preserving dimensionality

By default, any subsetting 2d data structures with a single number, single name, or a logical vector containing a single `TRUE` will simplify the returned output, i.e. it will return an object with lower dimensionality. To preserve the original dimensionality, you must use `drop = FALSE`

- For matrices and arrays, any dimensions with length 1 will be dropped:

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

str(a[1, , drop = FALSE])
#>  int [1, 1:2] 1 3
```

- Data frames with a single column will return just that column:

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[, "a"])
#>  int [1:2] 1 2

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

- Tibbles default to `drop = FALSE`, and `[` will never return a single vector.

The default `drop = TRUE` behaviour is a common source of bugs in functions: you check your code with a data frame or matrix with multiple columns, and it works. Six months later you (or someone else) uses it with a single column data frame and it fails with a mystifying error. When writing functions, get in the habit of always using `drop = FALSE` when subsetting a 2d object.

Factor subsetting also has a `drop` argument, but the meaning is rather different. It controls whether or not levels are preserved (not the dimensionality), and it defaults to `FALSE` (levels are preserved, not simplified by default). If you find you are using `drop = TRUE` a lot it's often a sign that you should be using a character vector instead of a factor.

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

4.2.6 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 the following code yield five missing values? (Hint: why is it different from `x[NA_real_]`?)

```
x <- 1:5
x[NA]
#> [1] NA NA NA NA NA
```

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 an 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?

4.3 Selecting a single element

There are two other subsetting operators: `[[` and `$`. `[[` is used for extracting single items, and `x$y` is a useful shorthand for `x[["y"]]`.

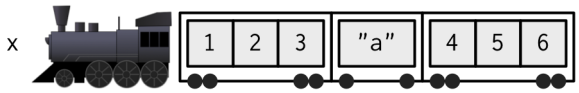
4.3.1 `[[`

`[[` is most important working with lists because subsetting a list with `[` always returns a smaller list. To help make this easier to understand we can use a metaphor:

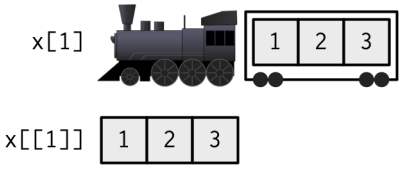
> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6."
>
> — @RLangTip, `https://twitter.com/RLangTip/status/268375867468681216`

Let's make a simple list and draw it as a train:

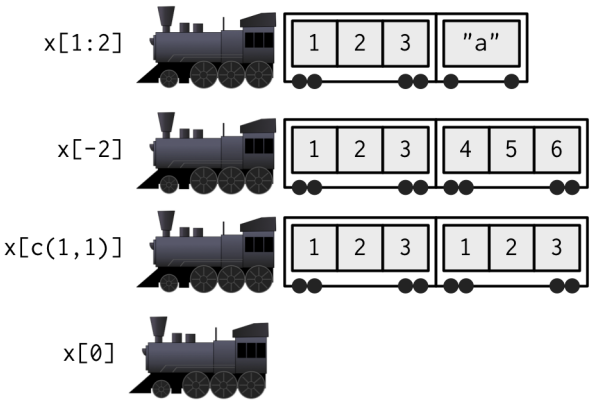
```
x <- list(1:3, "a", 4:6)
```



When extracting a single element, you have two options: you can create a smaller train, or you can extract the contents of a carriage. This is the difference between `[` and `[[`:



When extracting multiple elements (or zero!), you have to make a smaller train:



Because it can return only a single item, you must use `[[` with either a single positive integer or a string. If you use a vector with `[[`, it will subset recursively:

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

# Equivalent to
b[["a"]][["b"]][["c"]][["d"]]
#> [1] 1
```

`[[` is crucial for working with lists, but I recommend using it whenever you want your code to clearly express that it's working with a single item. That frequently arises in for loops, e.g., instead of writing:

```
for (i in 2:length(x)) {
  out[i] <- fun(x[i], out[i - 1])
}
```

It's better to write:

```
for (i in 2:length(x)) {
  out[[i]] <- fun(x[[i]], out[[i - 1]])
}
```

That reinforces to the reader that you expect to get and set individual values.

4.3.2 `$`

`$` is a shorthand operator: `x$y` is roughly equivalent to `x[["y"]]`. It's often used to access variables in a data frame, as in `mtcars$cyl` or `diamonds$carat`. One common mistake with `$` is to 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 4 4 4 4 8 8 8 4 4 4 8 6 8 4
```

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

```
x <- list(abc = 1)
x$a
#> [1] 1
x[["a"]]
#> NULL
```

To help avoid this behaviour I highly recommend setting the global option `warnPartialMatchDollar` to `TRUE`:

```
options(warnPartialMatchDollar = TRUE)
x$a
#> Warning in x$a: partial match of 'a' to 'abc'
#> [1] 1
```

(For data frames, you can also avoid this problem by using tibbles instead: they never do partial matching.)

4.3.3 Missing/out of bounds indices

It's useful to understand what happens with `[` and `[[` when you use an "invalid" index. The following tables summarise what happen when you subset a logical vector, list, and `NULL` with an out-of-bounds value (OOB), a missing value (i.e `NA_integer_`), and a zero-length object (like `NULL` or `logical()`) with `[` and `[[`. Each cell shows the result of subsetting the data structure named in the row by the type of index described in the column. I've only shown the results for logical vectors, but other atomic vectors behave similarly, returning elements of the same type.

`row[col]`	Zero-length	OOB	Missing
`NULL`	`NULL`	`NULL`	`NULL`
Logical	`logical(0)`	`NA`	`NA`
List	`list()`	`list(NULL)`	`list(NULL)`

With `[`, it doesn't matter whether the OOB index is a position or a name, but it does for `[[`:

`row[[col]]`	Zero-length	OOB (int)	OOB (chr)	Missing
`NULL`	`NULL`	`NULL`	`NULL`	`NULL`
Atomic	Error	Error	Error	Error
List	Error	Error	`NULL`	`NULL`

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

The inconsistency of the `[[` table above lead to the development of `purrr::pluck()` and `purrr::chuck()`. `pluck()` always returns `NULL` (or the value of the `.default` argument) when the element is missing; `chuck()` always throws an error:

`pluck(row, col)`	Zero-length	OOB (int)	OOB (chr)	Missing
`NULL`	`NULL`	`NULL`	`NULL`	`NULL`
Atomic	`NULL`	`NULL`	`NULL`	`NULL`
List	`NULL`	`NULL`	`NULL`	`NULL`

`chuck(row, col)`	Zero-length	OOB (int)	OOB (chr)	Missing
`NULL`	Error	Error	Error	Error
Atomic	Error	Error	Error	Error
List	Error	Error	Error	Error

The behaviour of `pluck()` makes it well suited for indexing into deeply nested data structures where the component you want does not exist always exist (as is common when working with JSON data from web APIs). `pluck()` also allows you to mingle integer and character indexes, and to provide an alternative default value if the item does not exist:

```
x <- list(
  a = list(1, 2, 3),
  b = list(3, 4, 5)
)

purrr::pluck(x, "a", 1)
#> [1] 1

purrr::pluck(x, "c", 1)
#> NULL

purrr::pluck(x, "c", 1, .default = NA)
#> [1] NA
```

4.3.4 `@` and `slot()`

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 S4.

4.3.5 Exercises

1. Brainstorm as manys way as possible to extract the third value from the `cyl` variable in the `mtcars` dataset.
2. 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)`)

4.4 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

# Duplicated indices go unchecked and may be problematic
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 structure of the original object. 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 `[[` + 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
```

4.5 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()`, `dplyr::arrange()`), but it is useful to understand how they are implemented with basic subsetting. This will allow you to adapt to new situations not handled by existing functions.

4.5.1 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        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"   "Male"

unname(lookup[x])
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"   "Male"
```

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

4.5.2 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`. An elegant way to do this is by combining `match()` and integer subsetting:

```
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
```

If you have multiple columns to match on, you'll need to first collapse them to a single column (with e.g. `interaction()`), but typically you are better off switching to a function design specifically for joining multiple tables like `merge()`, or `dplyr::left_join()`.

4.5.3 Random samples/bootstraps (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 accesses the values:

```
df <- data.frame(x = c(1, 2, 3, 1, 2), y = 5:1, z = letters[1:5])

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

# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 3 3 3 c
#> 2 2 4 b
#> 1 1 5 a

# Select 6 bootstrap replicates
df[sample(nrow(df), 6, replace = TRUE), ]
```

```
#>     x y z
#> 4   1 2 d
#> 4.1 1 2 d
#> 5   2 1 e
#> 1   1 5 a
#> 1.1 1 5 a
#> 2   2 4 b
```

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

4.5.4 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 3 3
#> 1 a 5 1
#> 2 b 4 2
#> 4 d 2 1
#> 5 e 1 2

df2[order(df2$x), ]
#>   z y x
```

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

You can sort vectors directly with `sort()`, or use `dplyr::arrange()` or similar to sort a data frame.

4.5.5 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
```

4.5.6 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 only 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
```

4.5.7 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`.

4.5.8 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] 2 5 8

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

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

```r
(x1 <- 1:10 %% 2 == 0)
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  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  TRUE
(y2 <- which(y1))
#> [1]  5 10

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

# X | Y <-> union(x, y)
x1 | y1
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  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 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 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. In more general cases, there are two important differences.

- When the logical vector contains `NA`, logical subsetting replaces these values by `NA` while `which()` drops these values. It's not uncommon to use `which()` for this side-effect, but that's
- `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.

4.5.9 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?

4.6 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")]`

5

Functions

5.1 Introduction

If you're reading this book, you've probably already created many R functions and know how to use them to reduce duplication in your code. In this chapter, you'll learn how to turn that informal, working knowledge into more rigorous, theoretical understanding. And while you'll see some interesting tricks and techniques along the way, keep in mind that what you'll learn here will be important for understanding the more advanced topics discussed later in the book.

Quiz

Answer the following questions to see if you can safely skip this chapter. You can find the answers in **??**.

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 usually 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 code throw an error when executed? 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. How do you ensure that cleanup action occurs regardless of how a function exits?

Outline

- Section **??** describes the basics of creating a function, the three main components of a function, and the exception to many function rules: primitive functions (which are implemented in C, not R).
- Section **??** shows you how R finds the value associated with a given name, i.e. the rules of lexical scoping.
- Section **??** is devoted to an important property of function arguments: they are only evaluated when used for the first time.
- Section **??** discusses the two primary ways that a function can exit, and how to define an exit handler, code that is run on exit, regardless of what triggers it.
- Section **??** shows you the various ways in which R disguises ordinary function calls, and how you can use the standard prefix form to better understand what's going on.

5.2 Function fundamentals

To understand functions in R you need to internalise two important ideas:

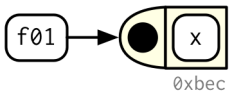
- Functions are objects, just as vectors are objects.
- Functions can be broken down into three components: arguments, body, and environment.

There are exceptions to every rule, and in this case, there is a small selection of "primitive" base functions that are implemented purely in C.

5.2.1 First-class functions

The most important thing to understand about R is that functions are objects in their own right, a language property often called "first-class functions". Unlike in many other languages, there is no special syntax for defining and naming a function: you simply create a function object (with `function`) and bind it to a name with `<-`:

```
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```



While you almost always create a function and then bind it to a name, the binding step is not compulsory. If you choose not to give a function a name, you get an **anonymous function**. This is useful when it's not worth the effort to figure out a name:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

A final option is to put functions in a list:

```
funs <- list(
  half = function(x) x / 2,
```

```
  double = function(x) x * 2
)

funs$double(10)
#> [1] 20
```

In R, you'll often see functions called **closures**. This name reflects the fact that R functions capture, or **enclose**, their environments.

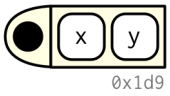
```
typeof(f01)
#> [1] "closure"
```

5.2.2 Function components

A function has three parts:

- The `formals()`, the list of arguments that control how you call the function.
- The `body()`, the code inside the function.
- The `environment()`, the data structure that determines how the function finds the values associated with the names.

I'll draw functions as in the following diagram. The black dot on the left is the environment. The two blocks to the right are the function arguments. I won't draw the body, because it's usually large, and doesn't help you understand the "shape" of the function.



While the formals and body are specified explicitly when you create a function, the environment is specified implicitly, based on *where* you defined the function. The function environment always exists, but it is only printed when the function isn't defined in the global environment.

```
f02 <- function(x) {
  # A comment
  x ^ 2
}

formals(f02)
```

```
#> $x

body(f02)
#> {
#>     x^2
#> }

environment(f02)
#> <environment: R_GlobalEnv>
```

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. It points to the source code used to create the function. The srcref is used for printing because, unlike `body()`, it contains code comments and other formatting.

```
attr(f02, "srcref")
#> function(x) {
#>   # A comment
#>   x ^ 2
#> }
```

5.2.3 Primitive functions

There is one exception to the rule that a function has three components. Primitive functions, like `sum()` and `[`, call C code directly.

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
`[`
#> .Primitive("[")
```

They have type "builtin" or "special":

```
typeof(sum)
#> [1] "builtin"
typeof(`[`)
#> [1] "special"
```

These functions exist primarily in C, not R, so their `formals()`, `body()`, and `environment()` are all `NULL`:

```
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Primitive functions are only found in the base package. While they have certain performance advantages, this benefit comes at a price: they are harder to write. For this reason, R-core generally avoids creating them unless there is no other option.

5.2.4 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. It's possible (although typically not useful) to call an anonymous function. Which of the two approaches below is correct? Why?

```
function(x) 3()
#> function(x) 3()
(function(x) 3)()
#> [1] 3
```

3. 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?

4. 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?

5. 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?

6. What are the three important components of a function?

7. When does printing a function not show the environment it was created in?

5.3 Lexical scoping

In Names and values, we discussed assignment, the act of binding a name to a value. Here we'll discuss **scoping**, the act of finding the value associated with a name.

The basic rules of scoping are quite intuitive, and you've probably already internalised them, even if you never explicitly studied them. For example, what will the following code return, 10 or 20?[1]

```
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

In this section, you'll learn the formal rules of scoping as well as some of its more subtle details. A deeper understanding of scoping will help you to use more advanced functional programming tools, and eventually, even to write tools that translate R code into other languages.

R uses **lexical scoping**[2]: it looks up the values of names based on how a function is defined, not how it is called. "Lexical" here is not the English

[1] I'll "hide" the answers to these challenges in the footnotes. Try solving them before looking at the answer; this will help you to better remember the correct answer. In this case, `g01()` will return `20`.

[2] Functions that automatically quote one or more arguments (sometimes called NSE functions) can override the default scoping rules to implement other varieties of scoping. You'll learn more about that in metaprogramming.

adjective "relating to words or a vocabulary". It's a technical CS term that tells us that the scoping rules use a parse-time, rather than a run-time structure.

R's lexical scoping follows four primary rules:

- Name masking
- Functions vs. variables
- A fresh start
- Dynamic lookup

5.3.1 Name masking

The following example illustrates the most basic principle of lexical scoping: names defined inside a function mask those defined outside of it.

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
#> [1] 1 2
```

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

```
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
#> [1] 2 1
```

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.[3]

[3] `g04()` returns `c(1, 2, 3)`.

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

The same rules apply to functions created by other functions, which we'll call **closures**. Closures will be described in more detail in [closures]; here we'll just look at how they interact with scoping. The following function, `g05()`, returns a function. What do you think this function will return when we call it?[4]

```
x <- 10
y <- 20

g05 <- function() {
  y <- 2
  function() {
    c(x, y)
  }
}
g06 <- g05()
g06()
```

This seems a little magical: how does R know what the value of `y` is after `j()` has returned? It works because `k` preserves the environment in which it was defined and because the environment includes the value of `y`. You'll learn more about how environments work in Environments.

5.3.2 Functions vs. variables

In R, functions are ordinary objects. This means the same scoping principles that apply to other objects also apply to functions:

```
g07 <- function(x) x + 1
g08 <- function() {
```

[4] `g06()` returns `c(10, 2)`.

```
  g07 <- function(x) x + 100
  g07(10)
}
g08()
#> [1] 110
```

The rule gets a little more complicated when a name is bound to a function and a non-function in different environments. When you use a name in function call, R will ignore non-function objects while looking for that value. For example, here `g9` takes on two different values:

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
#> [1] 110
```

But using the same name for two different things will make for confusing code, and is best avoided!

5.3.3 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?[5] (If you haven't seen `exists()` before: it returns `TRUE` if there's a variable of that name, otherwise it returns `FALSE`.)

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
g11()
```

[5] `g11()` returns 1 every time it is called.

You might be surprised that `g11()` always returns the same value. That happens because a new environment is created to host its execution every time a function is called. That means 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 **??**.)

5.3.4 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 differ depending on objects outside its environment:

```
g12 <- function() x + 1
x <- 15
g12()
#> [1] 16

x <- 20
g12()
#> [1] 21
```

This behaviour can be quite annoying. 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 to use `codetools::findGlobals()`. This function lists all the external dependencies (unbound symbols) within a function:

```
codetools::findGlobals(g12)
#> [1] "+" "x"
```

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

```
environment(g12) <- emptyenv()
g12()
#> Error in x + 1:
#>   could not find function "+"
```

Both of these approaches reveal why this undesirable behaviour exists: R

relies on lexical scoping to find *everything*, even the + operator. This provides a rather beautiful simplicity to R's scoping rules.

5.3.5 Exercises

1. What does the following code return? Why? Describe how each of the three c's is interpreted.

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

5.4 Lazy evaluation

In R, function arguments are **lazily evaluated**: they're only evaluated if accessed. For example, this code doesn't generate an error because x is never used:

```
h01 <- function(x) {
  10
}
```

```
h01(stop("This is an error!"))
#> [1] 10
```

This is an important feature because it allows you to do things like include potentially expensive computations in function arguments that will only be evaluated if needed.

5.4.1 Forcing evaluation

To **compel** the evaluation of an argument, use `force()`:

```
h02 <- function(x) {
  force(x)
  10
}
h02(stop("This is an error!"))
#> Error in force(x):
#>   This is an error!
```

It is usually not necessary to force evaluation. It's needed primarily for certain functional programming techniques which we'll cover in detail in function operators. Here, I want to show you the basic issue.

Take this small but surprisingly tricky function. It takes a single argument `x`, and returns a function that returns `x` when called.

```
capture1 <- function(x) {
  function() {
    x
  }
}
```

There's a subtle issue with this function: the value of `x` will be captured not when you call `capture()`, but when you call the function that `capture()` returns:

```
x <- 10
h03 <- capture1(x)
h04 <- capture1(x)

h03()
#> [1] 10
```

```
x <- 20
h04()
#> [1] 20
```

Even more confusingly this only happens once: the value is locked in after you have called `h03()`/`h04()` for the first time.

```
x <- 30
h03()
#> [1] 10
h04()
#> [1] 20
```

This behaviour is a consequence of lazy evaluation. The `x` argument is evaluated once `h03()`/`h04()` is called, and then its value is cached. We can avoid the confusion by forcing `x`:

```
capture2 <- function(x) {
  force(x)

  function() {
    x
  }
}

x <- 10
h05 <- capture2(x)

x <- 20
h05()
#> [1] 10
```

5.4.2 Promises

Lazy evaluation is powered by a data structure called a **promise**, or (less commonly) a thunk. We'll come back to this data structure in metaprogramming because it's one of the features of R that makes it most interesting as a programming language.

A promise has three components:

- The expression, like `x + y` which gives rise to the delayed computation.

- The environment where the expression should be evaluated.
- The value, which is computed and cached when the promise is first accessed by evaluating the expression in the specified environment.

The value cache ensures that accessing the promise multiple times always returns the same value. For example, you can see in the following code that `runif(1)` is only evaluated once:

```
h06 <- function(x) {
  c(x, x, x)
}

h06(runif(1))
#> [1] 0.0808 0.0808 0.0808
```

You can also create promises "by hand" using `delayedAssign()`:

```
delayedAssign("x", {print("Executing code"); runif(1)})
x
#> [1] "Executing code"
#> [1] 0.834
x
#> [1] 0.834
```

You'll see this idea again in advanced bindings.

5.4.3 Default arguments

Thanks to lazy evaluation, default value can be defined in terms of other arguments, or even in terms of variables defined later in the function:

```
h07 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h07()
#> [1]   1   2 110
```

Many base R functions use this technique, but I don't recommend it. It makes code harder to understand because it requires that you know exactly *when*

default arguments are evaluated in order to predict *what* they will evaluate to.

The evaluation environment is slightly different for default and user supplied arguments, as default arguments are evaluated inside the function. This means that seemingly identical calls can yield different results. It's easiest to see this with an extreme example:

```
h08 <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside f:
h08()
#> [1] "a" "x"

# ls() evaluated in global environment:
h08(ls())
#> [1] "f"
```

5.4.4 Missing arguments

If an argument has a default, you can determine if the value comes from the user or the default with `missing()`:

```
h09 <- function(x = 10) {
  list(missing(x), x)
}
str(h09())
#> List of 2
#>  $ : logi TRUE
#>  $ : num 10
str(h09(10))
#> List of 2
#>  $ : logi FALSE
#>  $ : num 10
```

`missing()` is best used sparingly. Take `sample()`, for example. How many arguments are required?

```
args(sample)
```

```
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

It looks like both `x` and `size` are required, but in fact `sample()` uses `missing()` to provide a default for `size` if it's not supplied. If I was to rewrite sample myself[6], I'd use an explicit `NULL` to indicate that `size` can be supplied, but it's not required:

```
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  if (is.null(size)) {
    size <- length(x)
  }

  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

You can make that pattern even simpler with a small helper. The infix `%||%` function uses the LHS if it's not null, otherwise it uses the RHS:

```
`%||%` <- function(lhs, rhs) {
  if (!is.null(lhs)) {
    lhs
  } else {
    rhs
  }
}

sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  size <- size %||% length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

Because of lazy evaluation, you don't need to worry about unnecessary computation: the RHS of `%||%` will only be evaluated if the LHS is null.

5.4.5 Exercises

1. What important property of `&&` make `x_ok()` work?

[6] Note that this only implements one way of calling `sample()`: you can also call it with a single integer, like `sample(10)`. This unfortunately makes `sample()` prone to silent errors in situations like `sample(x[i])`.

```
x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
#> [1] FALSE
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE
```

What is different with this code? Why is this behaviour undesirable here?

```
x_ok <- function(x) {
  !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
#> logical(0)
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE FALSE FALSE
```

2. The definition of `force()` is simple:

```
force
#> function (x)
#> x
#> <bytecode: 0x7faead84d3f8>
#> <environment: namespace:base>
```

Why is it better to `force(x)` instead of just `x`?

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

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

```
  x
}
f2()
```

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

```
y <- 10
f1 <- function(x = {y <- 1; 2}, y = 0) {
  c(x, y)
}
f1()
y
```

5. In `hist()`, the default value of `xlim` is `range(breaks)`, the default value for `breaks` is `"Sturges"`, and

```
range("Sturges")
#> [1] "Sturges" "Sturges"
```

   Explain how `hist()` works to get a correct `xlim` value.

6. Explain why this function works. Why is it confusing?

```
show_time <- function(x = stop("Error!")) {
  stop <- function(...) Sys.time()
  print(x)
}
show_time()
#> [1] "2018-09-02 01:04:04 EDT"
```

7. How many arguments are required when calling `library()`?

5.5 ... (dot-dot-dot)

Functions can have a special argument ... (pronounced dot-dot-dot). If a function has this argument, it can take any number of additional arguments. In other programming languages, this type of argument is often called a varargs, or the function is said to be variadic.

Inside a function, you can use ... to pass those additional arguments on to another function:

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
#> List of 2
#>  $ y: num 2
#>  $ z: num 3
```

It's possible (but rarely useful) to refer to elements of ... by their position, using a special form:

```
i03 <- function(...) {
  list(first = ..1, third = ..3)
}
str(i03(1, 2, 3))
#> List of 2
#>  $ first: num 1
#>  $ third: num 3
```

More often useful is `list(...)`, which evaluates the arguments and stores them in a list:

```
i04 <- function(...) {
  list(...)
}
str(i04(a = 1, b = 2))
#> List of 2
```

```
#>  $ a: num 1
#>  $ b: num 2
```

(See also `rlang::list2()` to support splicing and to silently ignore trailing commas, and `rlang::enquos()` to capture the unevaluated arguments, the topic of quasiquotation.)

There are two primary uses of ..., both of which we'll come back to later in the book:

- If your function takes a function as an argument, you want some way to pass on additional arguments to that function. In this example, `lapply()` uses ... to pass `na.rm` on to `mean()`:

```
x <- list(c(1, 3, NA), c(4, NA, 6))
str(lapply(x, mean, na.rm = TRUE))
#> List of 2
#>  $ : num 2
#>  $ : num 5
```

  We'll come back to this technique in Section **??**.

- If your function is an S3 generic, you need some way to allow methods to take arbitrary extra arguments. For example, take the `print()` function. There are different options for printing types of object, so there's no way for the print generic to prespecify every possible argument. Instead, it uses ... to allow individual methods to have different arguments:

```
print(factor(letters), max.levels = 4)

print(y ~ x, showEnv = TRUE)
```

  We'll come back to this use of ... in Section **??**.

Using ... comes with two downsides:

- When you use it to pass arguments on to another function, you have to carefully explain to the user where those arguments go. This makes it hard to understand the what you can do with functions like `lapply()` and `plot()`.
- Any misspelled arguments will not raise an error. This makes it easy for typos to go unnoticed:

```
sum(1, 2, NA, na_rm = TRUE)
#> [1] NA
```

. . . is a powerful tool, but be aware of the downsides.

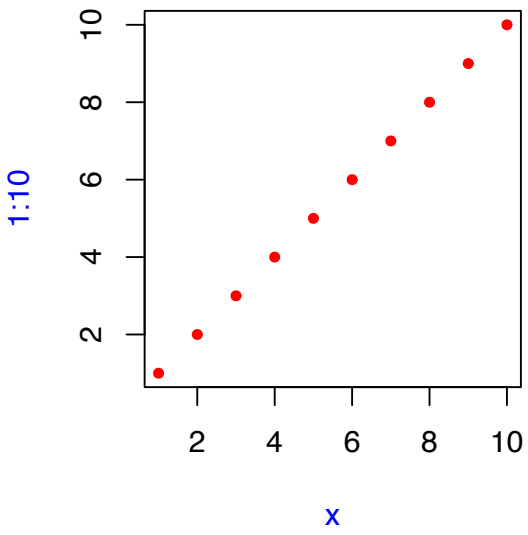
5.5.1 Exercises

1. Explain the following results:

```
sum(1, 2, 3)
#> [1] 6
mean(1, 2, 3)
#> [1] 1

sum(1, 2, 3, na.omit = TRUE)
#> [1] 7
mean(1, 2, 3, na.omit = TRUE)
#> [1] 1
```

2. In the following call, explain how to find the documentation for the named arguments in the following function call:

```
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```



3. Why does `plot(1:10, col = "red")` only colour the points, not the axes or labels? Read the source code of `plot.default()` to find out.

5.6 Exiting a function

Most functions exit in one of two ways[7]: either returning a value, indicating successful completion, or throwing an error, indicating failure. This section describes return values (implicit vs. explicit; visible vs. invisible), briefly discusses errors, and introduces exit handlers, which allow you to run code when a function exits, regardless of how it exits.

5.6.1 Implicit vs. explict returns

There are two ways that a function can return a value:

- Implicitly, where the last evaluated expression becomes the return value:

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
#> [1] 52
f(15)
#> [1] 452
```

- Explicitly, by calling `return()`:

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

[7]Functions can exit in other more esoteric ways like signalling a condition that is caught by an exiting handler, invoking a restart, or pressing "Q" in an interactive browser.

5.6.2 Invisible values

Most functions return visibly: calling the function in an interactive context causes the result to be automatically printed.

```
j03 <- function() 1
j03()
#> [1] 1
```

However, it's also possible to return an `invisible()` value, which is not automatically printed.

```
j04 <- function() invisible(1)
j04()
```

You can verify that the value exists either by explicitly printing it or by wrapping in parentheses:

```
print(j04())
#> [1] 1

(j04())
#> [1] 1
```

Alternatively, use `withVisible()` to return the value and a visibility flag:

```
str(withVisible(j04()))
#> List of 2
#>  $ value  : num 1
#>  $ visible: logi FALSE
```

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

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

And this is what makes it possible to chain assignment:

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

In general, any function called primarily for its side effects (like `<-`, `print()`,

or `plot()`) should return an invisible value (typically the value of the first argument).

5.6.3 Errors

If a function can not complete its assigned task, it should throw an error with `stop()`, which immediately terminates the execution of the function.

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
#> Error in j05():
#>   I'm an error
```

Errors indicate that something has gone wrong, and force the user to handle them. Some languages (like C, go, and rust) rely on special return values to indicate problems, but in R you should always throw an error. You'll learn more about errors, and how to handle them, in Conditions.

5.6.4 Exit handlers

Sometimes a function needs to make a temporary change to global state and you want to ensure those changes are restored when the function completes. It's painful to make sure you cleanup before any explicit return, and what happens if there's an error? Instead, you can set up an **exiting handler** that is called when the function terminates, regardless of whether it returns a value or throws an error.

To setup an exiting handler, call `on.exit()` with the code to be run. It will execute when the function exits, regardless of what causes it to exit:

```
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)

  if (x) {
    return(10)
  } else {
    stop("Error")
  }
```

```
}

f(TRUE)
#> [1] 4

f(FALSE)
#> [1] 2
```

Always set `add = TRUE` when using `on.exit()`. If you don't, each call to `on.exit()` will overwrite the previous exiting handler. Even when only registering a single handler, it's good practice to set `add = TRUE` so that you don't get an unpleasant surprise if you later add more exit handlers

`on.exit()` is important because it allows you to place clean-up actions next to actions with their cleanup operations.

```
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

When coupled with lazy evaluation, this leads to a very useful pattern for running a block of code in an altered environment:

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  force(code)
}

getwd()
#> [1] "/Users/chenghuang/Documents/Git/adv-r"
with_dir("~", getwd())
#> [1] "/Users/chenghuang"
```

See the withr package (`http://withr.r-lib.org`) for a collection of functions of this nature.

In R 3.4 and prior, `on.exit()` expressions are always run in the order in which they are created:

```
f <- function() {
  on.exit(message("a"), add = TRUE)
  on.exit(message("b"), add = TRUE)
}
f()
#> a
#> b
```

This can make cleanup a little tricky if some actions need to happen in a specific order; typically you want the most recent added expression to be run first. In R 3.5 and later, you can control this by setting `after = FALSE`:

```
f <- function() {
  on.exit(message("a"), add = TRUE, after = FALSE)
  on.exit(message("b"), add = TRUE, after = FALSE)
}
f()
#> b
#> a
```

5.6.5 Exercises

1. What does `load()` return? Why don't you normally see these values?
2. What does `write.table()` return? What would be more useful?
3. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?
4. 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).
5. 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, after = TRUE)
```

```
  sink(temp)
  on.exit(sink(), add = TRUE, after = 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?

5.7 Function forms

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."

— John Chambers

While everything that happens in R is a result of a function call, not all calls look the same. Function calls come in four varieties:

- In **prefix** form, the function name comes before its arguments, like `foofy(a, b, c)`. These constitute of the majority of function calls in R.
- In **infix** form, the function name comes inbetween its arguments, like `x + y`. Infix forms are used for many mathematical operators, as well as user-defined functions that begin and end with `%`.
- A **replacement** function assigns into what looks like a prefix function, like `names(df) <- c("a", "b", "c")`.
- **Special forms** like `[[`, `if`, and `for`, don't have a consistent structure and provide some of the most important syntax in R.

While four forms exist, you only need to use one, because any call can be written in prefix form. I'll demonstrate this property, and then you'll learn about each of the forms in turn.

5.7.1 Rewriting to prefix form

An interesting property of R is every infix, replacement, or special form can be rewritten in prefix form. Rewriting in prefix form is useful because it helps you better understand the structure of the language, and it gives you the real name of every function. Knowing the real name of non-prefix functions is useful because it allows you to modify them for fun and profit.

The following example shows three pairs of equivalent calls, rewriting an infix form, replacement form, and a special form into prefix form.

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

Knowing the function name of a non-prefix function allows you to override its behaviour. For example, if you're ever feeling particularly evil, run the following code while a friend is away from their computer. It will introduce a fun bug: 10% of the time, 1 will be added to any numeric calculation inside of parentheses.

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 4 3 3 3 3 3 3 3 4 3 3 3 3 4 3 3 3 3 3 3 3 4 3 3 3 4 3 3 3
#> [33] 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3
rm("(")
```

Of course, overriding built-in functions like this is a bad idea, but, as you'll learn about in [metaprogramming], it's possible to apply it only to selected code blocks. This provides a clean and elegant approach to writing domain specific languages and translators to other languages.

A more useful technique is to use this knowledge when using functional pro-

gramming tools. For example, you 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 more simply by relying on the existing `+` function:

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

We'll explore this idea in detail in functionals.

5.7.2 Prefix form {prefix-form}

The prefix form is the most common form in R code, and indeed in the majority of programming languages. Prefix calls in R are a little special because you can specify arguments in three ways:

- By position, like `help(mean)`.
- Using partial matching, like `help(to = mean)`.
- By name, like `help(topic = mean)`.

As illustrated by the following chunk, arguments are matched by exact name, then with unique prefixes, and finally by position.

```
k01 <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(k01(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(k01(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Can abbreviate long argument names:
```

```
str(k01(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(k01(1, 3, b = 1))
#> Error in k01(1, 3, b = 1):
#>   argument 3 matches multiple formal arguments
```

Generally, only 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 never use partial matching. See the tidyverse style guide, `http://style.tidyverse.org/syntax.html#argument-names`, for more advice.

5.7.3 Infix functions

Infix functions are so called because the function name comes **in**between its arguments, and hence infix functions have two arguments. R comes with a number of built-in infix operators: `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, and `<<-`. You can also create your own infix functions that start and end with `%`, and base R uses this to additionally define `%%`, `%*%`, `%/%`, `%in%`, `%o%`, and `%x%`.

Defining your own infix function is simple. You create a two argument function and bind it to a name that starts and ends with `%`:

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

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters except "%". 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)

"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 are two special infix functions that can be called with a single argument: `+` and `-`.

```
-1
#> [1] -1
+10
#> [1] 10
```

5.7.4 Replacement functions

Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`. They must have arguments named `x` and `value`, and 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
}
```

Replacement functions are used by placing the function call on the LHS of `<-`:

```
x <- 1:10
second(x) <- 5L
x
#>  [1]  1  5  3  4  5  6  7  8  9 10
```

I say they "act" like they modify their arguments in place, because, as discussed in Modify-in-place, they actually create a modified copy. We can see that by using `tracemem()`:

```
x <- 1:10
tracemem(x)
#> <0x7ffae71bd880>

second(x) <- 6L
#> tracemem[0x7ffae71bd880 -> 0x7ffae61b5480]:
#> tracemem[0x7ffae61b5480 -> 0x7ffae73f0408]: second<-
```

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

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

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

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

Combining replacement with other functions requires more complex translation. For example, this:

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

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

Is translated into:

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

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

5.7.5 Special forms

Finally, there are a bunch of language features that are usually written in special ways, but also have prefix forms. These include parentheses:

- `(x)` (`` `(`(x) ``)
- `{x}` (`` `{`(x) ``).

The subsetting operators:

- `x[i]` (`` `[`(x, i) ``)
- `x[[i]]` (`` `[[`(x, i) ``)

And the tools of control flow:

- `if (cond) true` (`` `if`(cond, true) ``)
- `if (cond) true else false` (`` `if`(cond, true, false) ``)
- `for(var in seq) action` (`` `for`(var, seq, action) ``)
- `while(cond) action` (`` `while`(cond, action) ``)
- `repeat expr` (`` `repeat`(expr) ``)
- `next` (`` `next`() ``)
- `break` (`` `break`() ``)

Finally, the most complex is the "function" function:

- `function(arg1, arg2) {body}` (`` `function`(alist(arg1, arg2), body, env) ``)

Knowing the name of the function that underlies the special form is useful for getting documentation. `?(` is a syntax error; `` ?`( `` will give you the documentation for parentheses.

Note that all special forms are implemented as primitive functions (i.e. in C); that means printing these functions is not informative:

```
`for`
#> .Primitive("for")
```

5.8 Invoking a function

Suppose you had a list of function arguments:

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

How could you then send that list to `mean()`? In base R, you need `do.call()`:

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

5.8.1 Exercises

1. Rewrite the following code snippets into prefix form:

```
1 + 2 + 3

1 + (2 + 3)

if (length(x) <= 5) x[[5]] else x[[n]]
```

2. 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)
```

3. Explain why the following code fails:

```
modify(get("x"), 1) <- 10
#> Error: target of assignment expands to non-language object
```

4. Create a replacement function that modifies a random location in a vector.
5. Write your own version of + that will paste its inputs together if they are character vectors but behaves as usual otherwise. In other words, make this code work:

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

```
"a" + "b"
#> [1] "ab"
```

6. Create a list of all the replacement functions found in the base package. Which ones are primitive functions? (Hint use `apropos()`)
7. What are valid names for user-created infix functions?
8. Create an infix `xor()` operator.
9. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`. You might call them `%n%`, `%u%`, and `%/%` to match conventions from mathematics.

5.9 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 infix and replacement functions.
7. You use `on.exit()`; see on exit for details.

6

Environments

6.1 Introduction

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 lexical scoping. Understanding environments is not necessary for day-to-day use of R. But they are important to understand because they power many important R features like lexical scoping, namespaces, and R6 classes, and interact with evaluation to give you powerful tools for making domain specific languages, like dplyr and ggplot2.

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 answers.

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

- Environment basics introduces you to the basic properties of an environment and shows you how to create your own.
- Recursing over environments provides a function template for computing with environments, illustrating the idea with a useful function.
- Explicit environments briefly discusses three places where environments are useful data structures for solving other problems.

Prerequisites

This chapter will use rlang functions for working with environments, because it allows us to focus on the essence of environments, rather than the incidental details.

```
library(rlang)

# Some API changes that haven't made it in rlang yet
search_envs <- function() {
  rlang:::new_environments(c(
    list(global_env()),
    head(env_parents(global_env()), -1)
  ))
}
```

Note that the `env_` functions in rlang are designed to work with the pipe: all take an environment as the first argument, and many also return an environment. I won't use the pipe in this chapter in the interest of keeping the code as simple as possible, but you should consider it for your own code.

6.2 Environment basics

Generally, an environment is similar to a named list, with four important exceptions:

- Every name must be unique.
- The names in an environment are not ordered (i.e. it doesn't make sense to ask what the first element of an environment is).

- An environment has a parent.
- Environments are not copied when modified.

Let's explore these ideas with code and pictures.

6.2.1 Basics

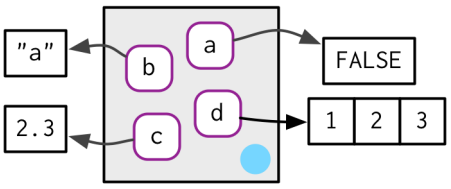
To create an environment, use `rlang::env()`. It works like `list()`, taking a set of name-value pairs:

```
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3,
)
```

In Base R

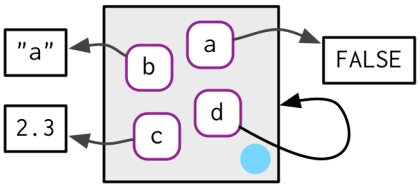
Use `new.env()` to creates a new environment. Ignore the `hash` and `size` parameters; they are not needed. Note that you can not simultaneously create and define values; use `$<-`, as shown below.

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, with no implied order (i.e. it doesn't make sense to ask which is the first element in an environment). For that reason, we'll draw the environment as so:



As discussed in names and values, environments have reference semantics: unlike most R objects, when you modify them, you modify them in place, and don't create a copy. One important implication is that environments can contain themselves. This means that environments go one step further in their level of recursion than lists: an enviroment can contain any object, including itself!

```
e1$d <- e1
```



Printing an evironment just displays its memory address, which is not terribly useful:

```
e1
#> <environment: 0x7f978d4608e8>
```

Instead, we'll use `env_print()` which gives us a little more information:

```
env_print(e1)
#> <environment: 0x7f978d4608e8>
#>   parent: <environment: global>
#>   bindings:
#>    * a: <lgl>
#>    * b: <chr>
#>    * c: <dbl>
#>    * d: <env>
```

You can use `env_names()` to get a character vector giving the current bindings

```
env_names(e1)
#> [1] "a" "b" "c" "d"
```

In Base R

In R 3.2.0 and greater, use `names()` to list the bindings in an environment. If your code needs to work with R 3.1.0 or earlier, use `ls()`, but note that the default value of `all.names` is `FALSE` so you don't see any bindings that start with `.`.

6.2.2 Important environments

We'll talk in detail about special environments in Special environments, but for now we need to mention two. The current environment, or `current_env()`

is the environment in which code is currently executing. When you're experimenting interactively, that's usually the global environment, or `global_env()`. The global environment is sometimes called your "workspace", as it's where all interactive (i.e. outside of a function) computation takes place.

Note that to compare environments, you need to use `identical()` and not `==`:

```
identical(global_env(), current_env())
#> [1] TRUE

global_env() == current_env()
#> Error in global_env() == current_env():
#>   comparison (1) is possible only for atomic and list types
```

In Base R

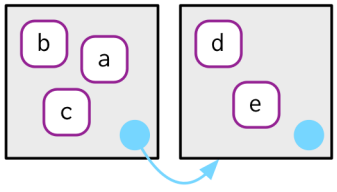
Access the global environment with `globalenv()` and the current environment with `environment()`. The global environment is printed as `Rf_GlobalEnv` and `.GlobalEnv`.

6.2.3 Parents

Every environment has a **parent**, another environment. In diagrams, the parent is shown as a small pale blue circle and arrow that points to another environment. The parent is what's used to implement lexical scoping: if a name is not found in an environment, then R will look in its parent (and so on).

You can set the parent environment by supplying an unnamed argument to `env()`. If you don't supply it, it defaults to the current environment.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```



We use the metaphor of a family to name environments relative to one another. The grandparent of an environment is the parent's parent, and the ancestors

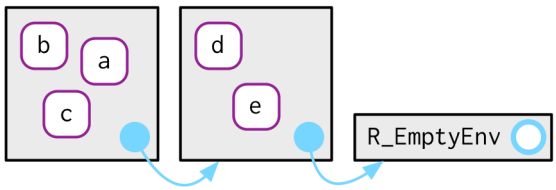
include all parent environments up to the empty environment. To save space, I typically won't draw all the ancestors; just remember whenever you see a pale blue circle, there's a parent environment somewhere.

You can find the parent of an environment with `env_parent()`:

```
env_parent(e2b)
#> <environment: 0x7f978f1c8a48>
env_parent(e2a)
#> <environment: R_GlobalEnv>
```

Only one environment doesn't have a parent: the **empty** environment. I draw the empty environment with a hollow parent environment, and where space allows I'll label it with `R_EmptyEnv`, the name R uses.

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
```



You'll get an error if you try to find the parent of the empty environment:

```
env_parent(empty_env())
#> Error: The empty environment has no parent
```

You can list all ancestors of an environment with `env_parents()`:

```
env_parents(e2b)
#> [[1]]  <env: 0x7f978f1c8a48>
#> [[2]] $ <env: global>

env_parents(e2d)
#> [[1]]  <env: 0x7f978e67d1b0>
#> [[2]] $ <env: empty>
```

By default, `env_parents()` continues until it hits either the global environment or the empty environment. You can control this behaviour with the `last` environment.

In Base R

Use `parent.env()` to find the parent of an environment. No base function returns all ancestors.

6.2.4 Getting and setting

You can get and set elements of an environment with `$` and `[[` in the same way as a list:

```
e3 <- env(x = 1, y = 2)
e3$x
#> [1] 1
e3$z <- 3
e3[["z"]]
#> [1] 3
```

But you can't use `[[` with numeric indices, and you can't use `[`:

```
e3[[1]]
#> Error in e3[[1]]:
#>   wrong arguments for subsetting an environment

e3[c("x", "y")]
#> Error in e3[c("x", "y")]:
#>   object of type 'environment' is not subsettable
```

`$` and `[[` will return `NULL` if the binding doesn't exist. Use `env_get()` if you want an error:

```
e3$xyz
#> NULL

env_get(e3, "xyz")
#> Error in env_get(e3, "xyz"):
#>   object 'xyz' not found
```

If you want to use a default value if the binding doesn't exist, you can use the `default` argument.

```
env_get(e3, "xyz", default = NA)
#> [1] NA
```

There are two other ways to add bindings to an environment:

- `env_poke()`[1] takes a name (as string) and a value:

```
env_poke(e3, "a", 100)
e3$a
#> [1] 100
```

- `env_bind()` allows you to bind multiple values:

```
env_bind(e3, a = 10, b = 20)
env_names(e3)
#> [1] "x" "y" "z" "a" "b"
```

You can determine if an environment has a binding with `env_has()`:

```
env_has(e3, "a")
#>    a
#> TRUE
```

Unlike lists, setting an element to `NULL` does not remove it. Instead, use `env_unbind()`:

```
e3$a <- NULL
env_has(e3, "a")
#>    a
#> TRUE

env_unbind(e3, "a")
env_has(e3, "a")
#>     a
#> FALSE
```

Unbinding a name doesn't delete the object. That's the job of the garbage collector, which automatically removes objects with no names binding to them. This process is described in more detail in GC.

[1] You might wonder why rlang has `env_poke()` instead of `env_set()`. This is for consistency: `_set()` functions return a modified copy; `_poke()` functions modify in place.

In Base R

See `get()`, `assign()`, `exists()`, and `rm()`. These are designed interactively for use with the current environment, so working with other environments is a little clunky. Also beware the `inherits` argument: it defaults to `TRUE` meaning that the base equivalents will inspect the supplied environment and all its ancestors.

6.2.5 Finalisers

TODO: Add something once rlang has an API. Also mention in data structures below

6.2.6 Advanced bindings

There are two more exotic variants of `env_bind()`:

- `env_bind_exprs()` creates **delayed bindings**, which are evaluated the first time they are accessed. Behind the scenes, delayed bindings create promises, so behave in the same way as function arguments.

```
env_bind_exprs(current_env(), b = {Sys.sleep(1); 1})

system.time(print(b))
#> [1] 1
#>    user  system elapsed
#>       0       0       1
system.time(print(b))
#> [1] 1
#>    user  system elapsed
#>   0.001   0.000   0.000
```

  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.

- `env_bind_fns()` creates **active bindings** which are re-computed every time they're accessed:

```
env_bind_fns(current_env(), z1 = function(val) runif(1))

z1
```

```
#> [1] 0.0808
z1
#> [1] 0.834
```

The argument to the function allows you to also override behaviour when the variable is set:

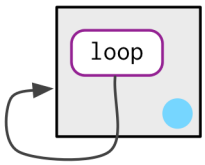
```
env_bind_fns(current_env(), z2 = function(val) {
  if (missing(val)) {
    2
  } else {
     stop("Don't touch z2!", call. = FALSE)
  }
})

z2
#> [1] 2
z2 <- 3
#> Error: Don't touch z2!
```

In Base R

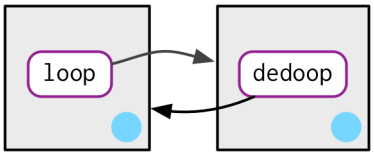
See `?delayedAssign()` and `?makeActiveBinding()`.

6.2.7 Exercises

1. List three ways in which an environment differs from a list.
2. Create an environment as illustrated by this picture.



3. Create a pair of environments as illustrated by this picture.



4. Explain why `e[[1]]` and `e[c("a", "b")]` don't make sense when `e` is an environment.

5. Create a version of `env_poke()` 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`).

6.3 Recursing over environments

If you want to operate on every ancestor of an environment, it's often convenient to write a recursive function. This section shows you how, applying your new knowledge of environments to write a function that given a name, finds the environment `where()` that name is defined, using R's regular scoping rules.

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 why `caller_env()` is a good default in calling environments.)

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    # Base case
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    # Success case
    env
  } else {
    # Recursive case
    where(name, env_parent(env))
  }
}
```

There are three cases:

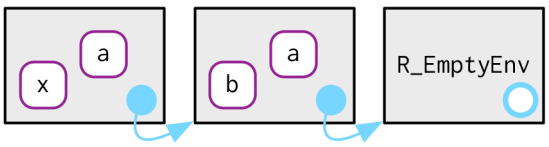
- The base case: we've reached the empty environment and haven't found the binding. We can't go any further, so we throw an error.
- The successful case: the name exists in this environment, so we return the environment.
- The recursive case: the name was not found in this environment, so try the parent.

These three cases are illustrated with these three examples:

```
where("yyy")
#> Error: Can't find yyy

x <- 5
where("x")
#> <environment: R_GlobalEnv>

where("mean")
#> <environment: base>
```

It might help to see a picture. Imagine you have two environments, as in the following code and diagram:

```
e4a <- env(empty_env(), a = 1, b = 2)
e4b <- env(e4a, x = 10, a = 11)
```



- `where(a, e4a)` will find `a` in `e4a`.
- `where("b", e4a)` doesn't find `b` in `e4a`, so it looks in its parent, `e4b`, and finds it there.
- `where("c", e4a)` looks in `e4a`, then `e4b`, then hits 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 = caller_env()) {
  if (identical(env, empty_env())) {
    # base case
```

```
  } else if (success) {
    # success case
  } else {
    # recursive case
    f(..., env = env_parent(env))
  }
}
```

Iteration vs recursion

It's possible to use a loop instead of recursion. I think it's harder to understand than the recursive version, but I include it because you might find it easier to see what's happening if you haven't written many recursive functions.

```
f2 <- function(..., env = caller_env()) {
  while (!identical(env, empty_env())) {
    if (success) {
      # success case
      return()
    }
    # inspect parent
    env <- env_parent(env)
  }

  # base case
}
```

6.3.1 Exercises

1. Modify `where()` to return *all* environments that contain a binding for `name`. Carefully think through what type of object the function will need to return.
2. 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.

6.4 Special environments

Most environments are not created by you (e.g. with `env()`) but are instead created by R. In this section, you'll learn about the most important environments, starting with the package environments. You'll then learn about the function environment bound to the function when it is created, and the (usually) ephemeral execution environment created every time the function is called. Finally, you'll see how the package and function environments interact to support namespaces, which ensure that a package always behaves the same way, regardless of what other packages the user has loaded.

6.4.1 Package environments and the search path

Each package attached by `library()` or `require()` becomes one of the parents of the global environment. The immediate parent of the global environment is the last package you attached[2]:

```
env_parent(global_env())
#> <environment: package:rlang>
#> attr(,"name")
#> [1] "package:rlang"
#> attr(,"path")
#> [1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/rlang"
```

And the parent of that package is the second to last package you attached:

```
env_parent(env_parent(global_env()))
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/stats"
```

If you follow all the parents back, you see the order in which every package has been attached. This is known as the **search path** because all objects in these environments can be found from the top-level interactive workspace.

[2]Note the difference between attached and loaded. A package is loaded automatically if you access one of its functions using `::`; it is only **attached** to the search path by `library()` or `require()`.

```
search_envs()
#>  [[1]] $ <env: global>
#>  [[2]] $ <env: package:rlang>
#>  [[3]] $ <env: package:stats>
#>  [[4]] $ <env: package:graphics>
#>  [[5]] $ <env: package:grDevices>
#>  [[6]] $ <env: package:utils>
#>  [[7]] $ <env: package:datasets>
#>  [[8]] $ <env: package:methods>
#>  [[9]] $ <env: Autoloads>
#> [[10]] $ <env: base>
```

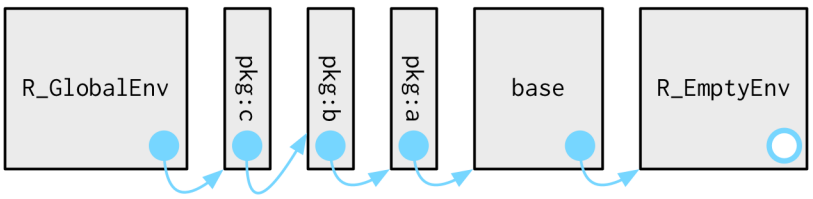
In Base R

You can access the names of the environments on the search path with `search()`

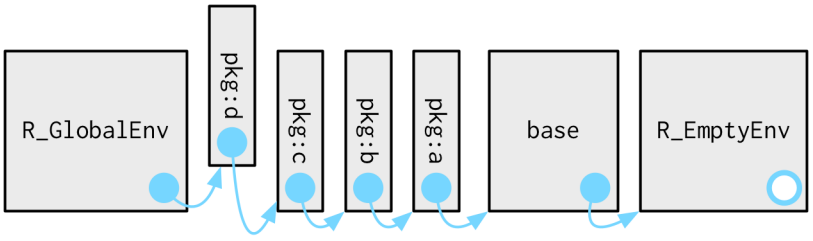
The last two environments on the search path are always the same:

- The `Autoloads` environment uses delayed bindings to save memory by only loading package objects (like big datasets) when needed.
- The base environment, `package:base` or sometimes just `base`, is the environment of the base package. It is special because it has to be able to bootstrap the loading of all other packages. You can access it directly with `base_env()`.

Graphically, the search path looks like this:



When you attach another package with `library()`, the parent environment of the global environment changes:



6.4.2 The function environment

A function binds the current environment when it is created. This is called the **function environment**, and is used for lexical scoping. Across computer languages, functions that capture their environments are called **closures**, which is why this term is often used interchangeably with function in R's documentation.

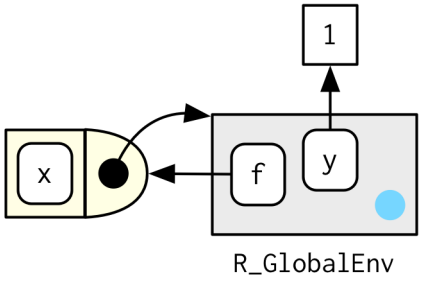
You can get the function environment with `fn_env()`:

```
y <- 1
f <- function(x) x + y
fn_env(f)
#> <environment: R_GlobalEnv>
```

In Base R

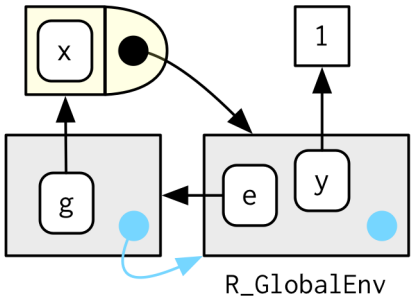
Use `environment(f)` to access the environment of function `f`.

In diagrams, I'll depict functions as rectangles with a rounded end that binds an environment.



In this case, `f()` binds the environment that binds the name `f` to the function. But that's not always the case: in the following example `g` is bound in a new environment `e`, but `g()` binds the global environment. The distinction between binding and being bound by is subtle but important; the difference is how we find `g` vs. how `g` finds its variables.

```
e <- env()
e$g <- function() 1
```



6.4.3 Namespaces

In the diagram above, you saw that the parent environment of a package varies based on what other packages have been loaded. This seems worrying: doesn't that mean that the package will find different functions if packages are loaded in a different order? The goal of **namespaces** is to make sure that this does not happen, and that every package works the same way regardless of what packages are attached by the user.

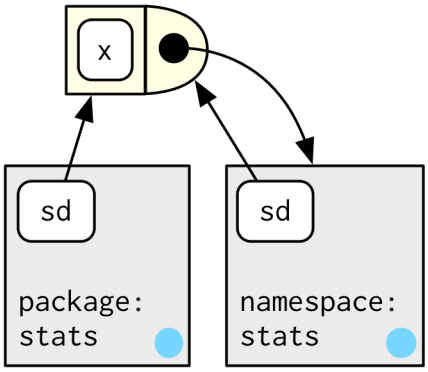
For example, take `sd()`:

```
sd
#> function (x, na.rm = FALSE)
#> sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
#>     na.rm = na.rm))
#> <bytecode: 0x7f978b1f0160>
#> <environment: namespace:stats>
```

`sd()` is defined in terms of `var()`, so you might worry that the result of `sd()` would be affected by any function called `var()` either in the global environment, or in one of the other attached packages. R avoids this problem by taking advantage of the function vs. binding environment described above. Every function in a package is associated with a pair of environments: the package environment, which you learned about earlier, and the **namespace** environment.

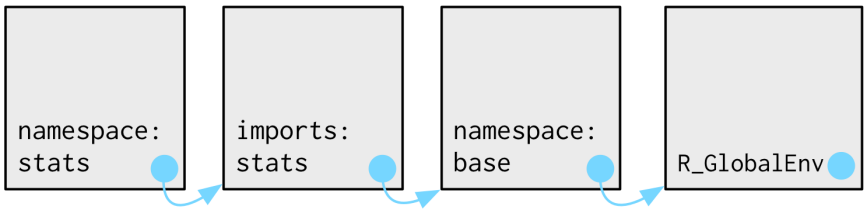
- The package environment is the external interface to the package. It's how you, the R user, find a function in an attached package or with `::`. Its parent is determined by search path, i.e. the order in which packages have been attached.
- The namespace environment is the internal interface to the package. The package environment controls how we find the function; the namespace controls how the function finds its variables.

Every binding in the package environment is also found in the namespace environment; this ensures every function can use every other function in the package. But some bindings only occur in the namespace environment. These are known as internal or non-exported objects, which make it possible to hide internal implementation details from the user.

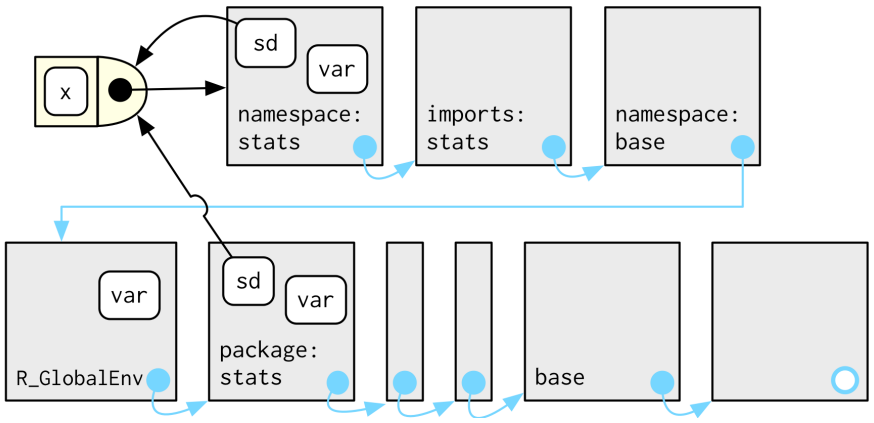


Every namespace environment has the same set of ancestors:

- Each namespace has an **imports** environment that contains bindings to all the functions used by the package. The imports environment is controlled by the package developer with the `NAMESPACE` file.
- Explicitly importing every base function would be tiresome, so the parent of the imports environment is the base **namespace**. The base namespace contains the same bindings as the base environment, but it has different parent.
- The parent of the base namespace is the global environment. This means that if a binding isn't defined in the imports environment the package will look for it in the usual way. This is usually a bad idea (because it makes code depend on other loaded packages), so `R CMD check` automatically warns about such code. It is needed primarily for historical reasons, particularly due to how S3 method dispatch works.



Putting all these diagrams together we get:



So when `sd()` looks for the value of `var` it always finds it in a sequence of environments determined by the package developer, but not by the package user. This ensures that package code always works the same way regardless of what packages have been attached by the user.

Note that there's no direct link between the package and namespace environments; the link is defined by the function environments.

6.4.4 Execution environments

The last important topic we need to cover is the **execution** environment. What will the following function return the first time it's run? What about the second?

```
g <- function(x) {
  if (!env_has(current_env(), "a")) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
```

Think about it for a moment before you read on.

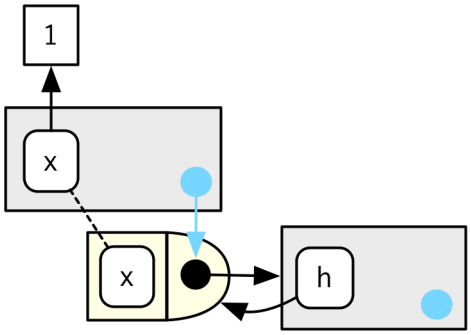
```
g(10)
#> Defining a
#> [1] 1
```

```
g(10)
#> Defining a
#> [1] 1
```

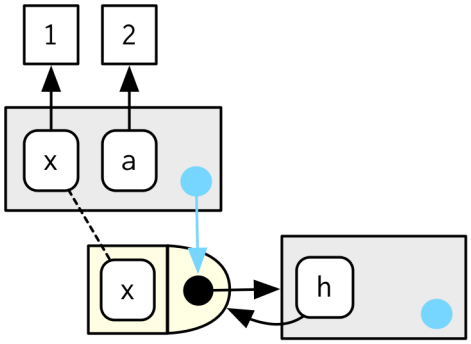
This function returns the same value every time because of the fresh start principle, described in a fresh start. Each time a function is called, a new environment is created to host execution. This is called the execution environment, and its parent is the function environment. Let's illustrate that process with a simpler function. I'll draw execution environments with an indirect parent; the parent environment is found via the function environment.

```
h <- function(x) {
  # 1.
  a <- 2 # 2.
  x + a
}
y <- h(1) # 3.
```

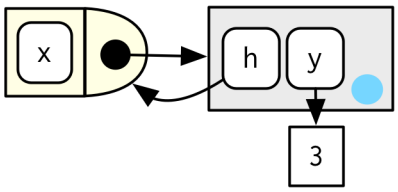
1. Function called with x = 1



2. a bound to value 2



3. Function completes returning value 3.
Execution environment goes away.



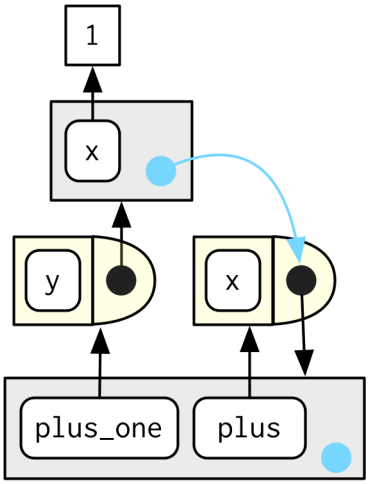
An execution environment is usually ephemeral; once the function has completed, the environment will be GC'd. There are several ways to make it stay around for longer. The first is to explicitly return it:

```
h2 <- function(x) {
  a <- x * 2
  current_env()
}

e <- h2(x = 10)
env_print(e)
#> <environment: 0x7f978f1e9f58>
#>   parent: <environment: global>
#>   bindings:
#>    * a: <dbl>
#>    * x: <dbl>
fn_env(h2)
#> <environment: R_GlobalEnv>
```

Another way to capture it is to return an object with a binding to that environment, like a function. The following example illustrates that idea with a function factory, `plus()`. We use that factory to create a function called `plus_one()`.

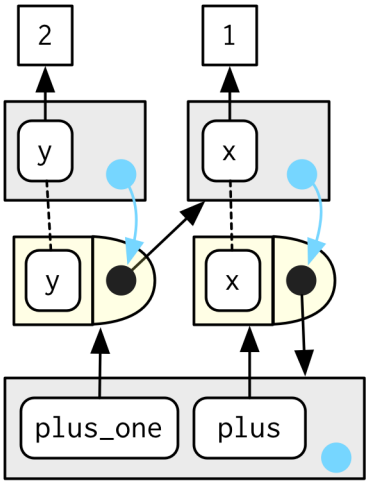
There's a lot going on in the diagram because the enclosing environment of `plus_one()` is the execution environment of `plus()`.

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

plus_one <- plus(1)
plus_one
#> function(y) x + y
#> <environment: 0x7f97903c0460>
```



What happens when we call `plus_one()`? Its execution environment will have the captured execution environment of `plus()` as its parent:

```
plus_one(2)
#> [1] 3
```



You'll learn more about function factories in functional programming.

6.4.5 Exercises

1. How is `search_envs()` different to `env_parents(global_env())`?

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. 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.

6.5 The call stack

There is one last environment we need to explain, the **caller** environment, accessed with `rlang::caller_env()`. This provides the environment from which the function was called, and hence varies based on how the function is called, not how the function was created. As we saw above this is a useful default whenever you write a function that takes an environment as an argument.

In Base R

`parent.frame()` is equivalent to `caller_env()`; just note that it returns an environment, not a frame.

To fully understand the caller environment we need to discuss two related concepts: the **call stack**, which is made up of **frames**. Executing a function creates two types of context. You've learned about one already: the execution environment is a child of the function environment, which is determined by where the function was created. There's another type of context created by where the function was called: this is called the call stack.

There are also a couple of small wrinkles when it comes to custom evaluation. See environments vs. frames for more details.

6.5.1 Simple call stacks

Let's illustrate this with a simple sequence of calls: `f()` calls `g()` calls `h()`.

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  stop()
}
```

The way you most commonly see a call stack in R is by looking at the `traceback()` after an error has occured:

```
f(x = 1)
#> Error:
traceback()
#> 4: stop()
#> 3: h(x = 3)
#> 2: g(x = 2)
#> 1: f(x = 1)
```

Instead of `stop()` + `traceback()` to understand the call stack, we're going to use `lobstr::cst()` to print out the **c**all **s**tack **t**ree:

```
h <- function(x) {
  lobstr::cst()
}
f(x = 1)
#>
#>   f(x = 1)
#>     g(x = 2)
#>       h(x = 3)
#>         lobstr::cst()
```

This shows us that `cst()` was called from `h()`, which was called from `g()`, which was called from `f()`. Note that the order is the opposite from `traceback()`. As the call stacks get more compliated, I think it's easier to understand the sequence of calls if you start from the beginning, rather than the end (i.e. `f()` calls `g()`; rather than `g()` was called by `f()`).

6.5.2 Lazy evaluation

The call stack above is simple - while you get a hint that there's some tree-like structure involved, everything happens on a single branch. This is typical of a call stack when all arguments are eagerly evaluated.

Let's create a more complicated example that involves some lazy evaluation. We'll create a sequence of functions, `a()`, `b()`, `c()`, that pass along an argument `x`.

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x

a(f())
#>
#>   a(f())
#>     b(x)
#>       c(x)
#>   f()
#>     g(x = 2)
#>       h(x = 3)
#>         lobstr::cst()
```

`x` is lazily evaluated so this tree gets two branches. In the first branch `a()` calls `b()`, then `b()` calls `c()`. The second branch starts when `c()` evaluates its argument `x`. This argument is evaluated in a new branch because the environment in which it is evaluated is the global environment, not the environment of `c()`.

6.5.3 Frames

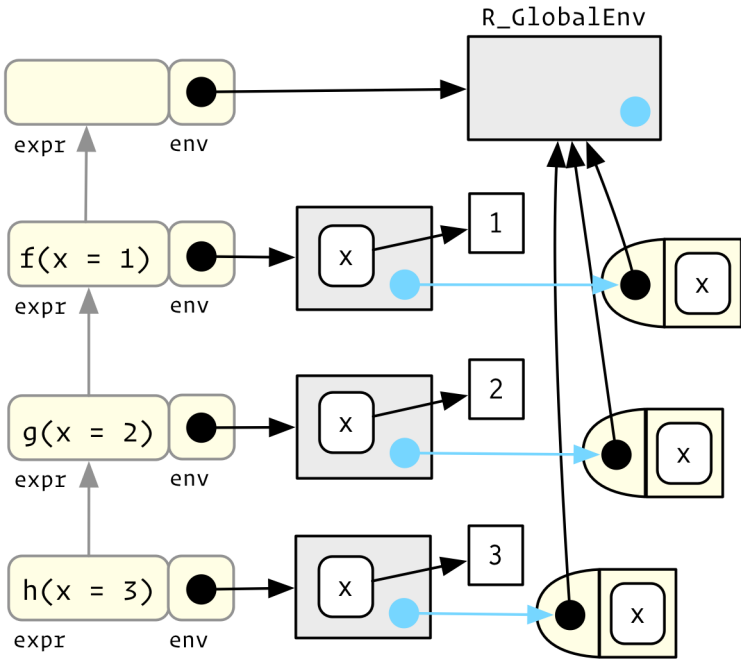
Each element of the call stack is a **frame**[3], also known as an evaluation context. The frame is an extremely important internal data structure, and R code can only access a small part of the data structure because it's so critical. A frame has three main components that are accessible from R:

- An expression (labelled with `expr`) giving the function call. This is what `traceback()` prints out.
- An environment (labelled with `env`), which is typically the execution envi-

[3]NB: `?environment` uses frame in a different sense: "Environments consist of a *frame*, or collection of named objects, and a pointer to an enclosing environment.". We avoid this sense of frame, which comes from S, because it's very specific and not widely used in base R. For example, the "frame" in `parent.frame()` is an execution context, not a collection of named objects.

ronment of a function. There are two main exceptions: the environment of the global frame is the global environment, and calling `eval()` also generates frames, where the environment can be anything.

- A parent, the previous call in the call stack (shown by a grey arrow).



(To focus on the calling environments, I have omitted the bindings in the global environment from `f`, `g`, and `h` to the respective function objects.)

The frame also holds exit handlers created with `on.exit()`, restarts and handlers for the condition system, and which context to `return()` to when a function completes. These are important for the internal operation of R, but are not directly accessible.

6.5.4 Dynamic scope

Looking up variables in the calling stack 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 the context in which it was called. Dynamic scoping is primarily useful for developing functions that

aid interactive data analysis. It is one of the topics discussed in non-standard evaluation.

6.5.5 Exercises

1. Write a function that lists all the variables defined in the environment in which it was called. It should return the same results as `ls()`.

6.6 As data structures

As well as powering scoping, environments are also useful data structures in their own right because they have reference semantics. There are three common problems that they can help solve:

- **Avoiding copies of large data**. Since environments have reference semantics, you'll never accidentally create a copy. This makes it a useful vessel for large objects. Bare environments are not that pleasant to work with; I recommend using R6 objects instead. Learn more in R6.
- **Managing state within a package**. 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 on exit).

- **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.

6.7 <<-

The ancestors of an environment have an important relationship to <<-. 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.

```
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 Closures.

6.7.1 Exercises

1. What does this function do? How does it differ from <<- and why might you prefer it?

```
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
    rebind(name, value, env_parent(env))
```

```
  }
}
rebind("a", 10)
#> Error: Can't find `a`
a <- 5
rebind("a", 10)
a
#> [1] 10
```

6.8 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 `caller_env()` or `parent.frame()`.
5. `<-` always creates a binding in the current environment; `<<-` rebinds an existing name in a parent of the current environment.

7

Conditions

7.1 Introduction

The **condition** system provides a paired set of tools that allow the author of a function to indicate that something unusual is happening, and the user of that function to deal with it. The function author **signals** conditions with functions like `stop()` (for errors), `warning()` (for warnings), and `message()` (for messages), then the function user can handle them with functions like `tryCatch()` and `withCallingHandlers()`. Understanding the condition system is important because you'll often need to play both roles: signalling conditions from the functions you create, and handle conditions signalled by the functions you call.

R offers a very powerful condition system based on ideas from Common Lisp. Like R's approach to object oriented programming, it is rather different to currently popular programming languages so it is easy to misunderstand, and there has been relatively little written about how to use it effectively. Historically, this has lead to few people (including me!) taking full advantage of its power. The goal of this chapter is to remedy that situation. Here you will learn about the big ideas of R's conditional system, as well as learning a bunch of practical tools that will make your code stronger.

I found two resources particularly useful when writing this chapter. You may also want to read them if you want to learn more about the inspirations and motivations for the system:

- *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`.

I also found it helpful to work through the underlying C code that implements these ideas. If you're interested in understanding how it all works, you might find my notes (`https://gist.github.com/hadley/4278d0a6d3a10e42533d59905fbed0ac`) to be useful.

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 answers.

1. What are the three most important types of condition?
2. What function do you use to ignore errors in block of code?
3. What's the main difference between `tryCatch()` and `withCallingHandlers()`?
4. Why might you want to create a custom error object?

Overview

- Signalling conditions introduces the basic tools for signalling conditions, and discusses when it is appropriate to use each type.
- Ignoring conditions teaches you about the simplest tools for handling conditions: functions like `try()` and `supressMessages()` that swallow conditions and prevent them from getting to the top level.
- Handling conditions introduces the condition **object**, and the two fundamental tools of condition handling: `tryCatch()` for error conditions, and `withCallingHandlers()` for everything else.
- Custom conditions shows you how to extend the built-in condition objects to store useful data that condition handlers can use to make more informed decisions.
- Applications closes out the chapter with a grab bag of practical applications based on the low-level tools found in earlier sections.

7.1.1 Prerequisites

As well as base R functions, this chapter uses condition signalling and handling functions from rlang.

```
library(rlang)
```

7.2 Signalling conditions

There are three conditions that you can signal in code: errors, warnings, and messages.

- Errors are the most severe; they indicate that there is no way for a function to continue and execution must stop.
- Messages are the mildest; they are way of informing the user that some action has been performed on their behalf.
- Warnings fall somewhat in between, and typically indicate that something has gone wrong but the function has been able to at least partially recover.

There is a final condition that can only be generated interactively: an interrupt, which indicates that the user has "interrupted" execution by pressing Escape, Ctrl + Break, or Ctrl + C (depending on the platform).

Conditions are usually displayed prominently, in a bold font or coloured red, depending on the R interface. You can tell them apart because errors always start with "Error", warnings with "Warning message", and messages with nothing.

```
stop("This is what an error looks like")
#> Error in eval(expr, envir, enclos):
#>   This is what an error looks like

warning("This is what a warning looks like")
#> Warning: This is what a warning looks like

message("This is what a message looks like")
#> This is what a message looks like
```

The following three sections describe errors, warnings, and message in more detail.

7.2.1 Errors

In base R, errors are signalled, or **thrown**, by `stop()`:

```
f <- function() g()
g <- function() h()
h <- function() stop("This is an error!")

f()
#> Error in h():
#>   This is an error!
```

By default, the error message includes the call, but this is typically not useful (and recapitulates information that you can easily get from `traceback()`), so I think it's good practice to use `call. = FALSE`:

```
h <- function() stop("This is an error!", call. = FALSE)
f()
#> Error: This is an error!
```

The rlang equivalent to `stop()`, `rlang::abort()`, does this automatically. We'll use `abort()` throughout this chapter, but we won't get to its most compelling feature, the ability to add additional metadata to the condition object, until we're near the end of the chapter.

```
h <- function() abort("This is an error!")
f()
#> Error: This is an error!
```

(Note that `stop()` pastes together multiple inputs, while `abort()` does not. To create complex error messages with abort, I recommend using `glue::glue()`. This allows us to use other arguments to `abort()` for useful features that you'll learn about in custom conditions.)

The best error messages tell you what is wrong and point you in the right direction to fix the problem. Writing good error messages is hard because errors usually occur when the user has a flawed mental model of the function. As a developer, it's hard to imagine how the user might be thinking incorrectly about your function, and thus it's hard to write a message that will steer the user in the correct direction. That said, the tidyverse style guide discusses a few general principles that we have found useful: `http://style.tidyverse.org/error-messages.html`.

7.2.2 Warnings

Warnings, signalled by `warning()`, are weaker than errors: they signal that something has gone wrong, but the code has been able to recover and continue. Unlike errors, you can have multiple warnings from a single function call:

```
fw <- function() {
  cat("1\n")
  warning("W1")
  cat("2\n")
  warning("W2")
  cat("3\n")
  warning("W3")
}
```

By default, warnings are cached and printed only when control returns to the top level:

```
fw()
#> 1
#> 2
#> 3
#> Warning messages:
#> 1: In f() : W1
#> 2: In f() : W2
#> 3: In f() : W3
```

You can control this behaviour with the `warn` option:

- To make warnings appear immediately, set `options(warn = 1)`.
- To turn warnings into errors, set `options(warn = 2)`. This is usually the easiest way to debug a warning, as once it's an error you can use tools like `traceback()` to find the source.
- Restore the default behaviour with `option(warn = 0)`.

Like `stop()`, `warning()` also has a call argument. It is slightly more useful (since warnings are often more distant from their source), but I still generally suppress it with `call. = FALSE`. Like `rlang::abort()`, the rlang equivalent of `warning()`, `rlang::warn()`, also suppresses the `call.` by default.

Warnings occupy a somewhat challenging place between messages ("you should know about this") and errors ("you must fix this!"), and it's hard to give precise advice on when to use them. Generally, be restrained, as warnings are easy to miss if there's a lot of other output, and you don't want your function to recover too easily from clearly invalid input. In my opinion, base

R tends to overuse warnings, and many warnings in base R would be better off as errors. For example, I think these warnings would be more helpful as errors:

```
formals(1)
#> Warning in formals(fun): argument is not a function
#> NULL

file.remove("this-file-doesn't-exist")
#> Warning in file.remove("this-file-doesn't-exist"): cannot remove file
#> 'this-file-doesn't-exist', reason 'No such file or directory'
#> [1] FALSE

lag(1:3, k = 1.5)
#> Warning in lag.default(1:3, k = 1.5): 'k' is not an integer
#> [1] 1 2 3
#> attr(,"tsp")
#> [1] -1  1  1
```

There only a couple of cases where using a warning is clearly appropriate:

- When you **deprecate** a function you want to allow older code to continue to work (so ignoring the warning is ok) but you want to encourage the user to switch to a new function.
- When you are reasonably certain you can recover from a problem: If you were 100% certain that you could fix the problem, you wouldn't need any message; if you were more uncertain that you could correctly fix the issue, you'd throw an error.

Otherwise use warnings with restraint, and carefully consider if an error would be more appropriate.

7.2.3 Messages

Messages, signalled by `message()`, are informational; use them to tell the user that you've done something on their behalf. Good messages are a balancing act: you want to provide just enough information so the user knows what's going on, but not so much that they're overwhelmed.

`messages()` are displayed immediately and do not have a `call.` argument:

```
fm <- function() {
  cat("1\n")
  message("M1")
```

```
  cat("2\n")
  message("M2")
  cat("3\n")
  message("M3")
}

fm()
#> 1
#> M1
#> 2
#> M2
#> 3
#> M3
```

Good places to use a message are:

- When a default argument requries some non-trivial amount of computation and you want to tell the user what value was used. For example, ggplot2 reports the number of bins used if you don't supply a `binwidth`.
- In functions that are called primarily for their side-effects which would otherwise be silent. For example, when writing files to disk, calling a web API, or writing to a database, it's useful provide regular status messages telling the user what's happening.
- When you're about to start a long running process with no intermediate output. A progress bar (e.g. with progress (`https://github.com/r-lib/progress`)) is better, but a message is a good place start.
- When writing a package, you sometimes want to display a message when your package is loaded (i.e. in `.onAttach()`); here you must use `packageStartupMessage()`.

Generally any function that produces a message should have some way to suppress it, like a `quiet = TRUE` argument. It is possible to suppress all messages with `suppressMessages()`, as you'll learn shortly, but it is nice to also give finer grain control.

It's important to compare `message()` to the closely related `cat()`. In terms of usage and result, they appear quite similar[1]:

```
cat("Hi!\n")
#> Hi!
```

[1] But note that `cat()` requires an explicit trailing `"\n"` to print a new line.

```
message("Hi!")
#> Hi!
```

However, the *purposes* of `cat()` and `message()` are different. Use `cat()` when the primary role of the function is to print to the console, like `print()` or `str()` methods. Use `message()` as a side-channel to print to the console when the primary purpose of the function is something else. In other words, `cat()` is for when the user *asks* for something to be printed and `message()` is for when developer *elects* to print something.

7.2.4 Exercises

1. Write a wrapper around `file.remove()` that throws an error if the file to be deleted does not exist.
2. What does the `appendLF` argument to `message()` do? How is it related to `cat()`?
3. What does `options(error = recover)` do? Why might you use it?
4. What does `options(error = quote(dump.frames(to.file = TRUE)))` do? Why might you use it?

7.3 Ignoring conditions

The simplest way of handling conditions in R is to simply ignore them:

- Ignore errors with `try()`.
- Ignore warnings with `suppressWarnings()`.
- Ignore messages with `suppressMessages()`.

These functions are heavy handed as you can't use them to suppress a single type of condition that you know about, while allowing everything else to pass through. We'll come back to that challenge later in the chapter.

`try()` allows execution to continue even after an error has occurred. 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 displayed[2] 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
```

It is possible, but not recommended, to save the result of `try()` and perform different actions based on whether or not the code succeed or failed[3]. Instead, it is better to use `tryCatch()` or a higher-level helper; you'll learn about those shortly. A simple, but useful, pattern is to do assignment inside the call: this lets you define a default value to be used if the code does not succeed.

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

`suppressWarnings()` and `suppressMessages()` suppress all warnings and messages. Unlike errors, messages and warnings don't terminate execution, so there maybe multiple signalled in a single block.

```
suppressWarnings({
  warning("Uhoh!")
  warning("Another warning")
  1
})
#> [1] 1

suppressMessages({
  message("Hello there")
  2
```

[2] You can suppress the message with `try(..., silent = TRUE)`.

[3] You can tell if the expression failed because the result will have class `try-error`.

```
})
#> [1] 2

suppressWarnings({
  message("You can still see me")
  3
})
#> You can still see me
#> [1] 3
```

7.4 Handling conditions

Every condition has default behaviour: errors stop execution and return to the top level, warnings are captured and displayed in aggregate, and messages are immediately displayed. Condition **handlers** allow us to temporarily override or supplement the default behaviour.

Two functions, `tryCatch()` and `withCallingHandlers()`, allow us to register handlers, functions that take the signalled condition as their single argument. The registration functions have the same basic form:

```
tryCatch(
  error = function(cnd) {
    # code to run when error is thrown
  },
  code_to_run_while_handlers_are_active
)

withCallingHandlers(
  warning = function(cnd) {
    # code to run when warning is signalled
  },
  message = function(cnd) {
    # code to run when warning is signalled
  },
  code_to_run_while_handlers_are_active
)
```

They differ in the type of handlers that they create:

- `tryCatch()` defines **exiting** handlers; after the condition is handled, control returns to the context where `tryCatch()` was called. This makes `tryCatch()` most suitable for working with errors and interrupts, as these have to exit anyway.
- `withCallingHandlers()` defines **calling** handlers; after the condition is captured control returns to the context where the condition was signalled. This makes it most suitable for working with non-error conditions.

But before we can learn about and use these handlers, we need to talk a little bit about condition **objects**. These are created implicitly whenever you signal a condition, but become explicit inside the handler.

7.4.1 Condition objects

So far we've just signalled conditions, and not looked at the objects that are created behind the The easiest way to see a condition object is to catch one from a signalled condition. That's the job of `rlang::catch_cnd()`:

```
cnd <- catch_cnd(abort("An error"))
str(cnd)
#> List of 2
#>  $ message: chr "An error"
#>  $ call   : NULL
#>  - attr(*, "class")= chr [1:3] "rlang_error" "error" "condition"
```

Built-in conditions are lists with two elements:

- `message`, a length-1 character vector containing the text display to a user. To extract the message, use `conditionMessage(cnd)`.
- `call`, the call which triggered the condition. As described above, we don't use the call, so it will often be `NULL`. To extract it, use `conditionCall(cnd)`.

Custom conditions may contain other components, which we'll discuss in custom conditions.

Conditions also have a `class` attribute, which makes them S3 objects. We won't disucss S3 until S3, but fortunately, even if you don't know about S3, condition objects are quite simple. The most important thing to know know is that the `class` attribute is a character vector, and it determines which handlers will match the condition.

7.4.2 Exiting handlers

`tryCatch()` registers exiting handlers, and is typically used to handle error conditions. It allows you to override the default error behaviour. For example, the following code will return `10` instead of display an error:

```
tryCatch(
  error = function(cnd) 10,
  stop("This is an error!")
)
#> [1] 10
```

If no conditions are signalled, or the class of the signalled condition does not match the handler name, the code executes normally:

```
tryCatch(
  error = function(cnd) 10,
  1 + 1
)
#> [1] 2

tryCatch(
  error = function(cnd) 10,
  {
    message("Hi!")
    1 + 1
  }
)
#> Hi!
#> [1] 2
```

The handlers set up by `tryCatch()` are called **exiting** handlers because after the condition is signalled, control passes to the handler and never returns to the original code, effectively meaning that the code "exits":

```
tryCatch(
  message = function(cnd) "There",
  {
    message("Here")
    stop("This code is never run!")
  }
)
#> [1] "There"
```

Note that the code is evaluated in the environment of `tryCatch()`, but the handler code is not, because the handlers are functions. This is important to remember if you're trying to modify objects in the parent environment.

The handler functions are called with a single argument, the condition object. I call this argument `cnd`, by convention. This value is only moderately useful for the base conditions because they contain relatively little data. It's more useful when you make your own custom conditions, as you'll see shortly.

```
tryCatch(
  error = function(cnd) {
    paste0("--", conditionMessage(cnd), "--")
  },
  stop("This is an error")
)
#> [1] "--This is an error--"
```

`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, like deleting files, or closing connections. This is functionally equivalent to using `on.exit()` (and indeed that's how it's implemented) but it can wrap smaller chunks of code than an entire function.

7.4.3 Calling handlers

The handlers set up by `tryCatch()` are called exiting handlers, because they cause code to exit once the condition has been caught. By contrast, `withCallingHandler()` sets up **calling** handlers: code execution continues normally once the handler returns. This tends to make `withCallingHandlers()` a more natural pairing with the non-error conditions.

Compare the results of `tryCatch()` and `withCallingHandlers()` in the example below. The message are not printed in the first case, because the code is terminated once the exiting handler completes. They are printed in the second case, because a calling handler does not exit.

```
tryCatch(
  message = function(cnd) cat("Caught a message!\n"),
  {
    message("Someone there?")
    message("Why, yes!")
  }
)
```

```
#> Caught a message!

withCallingHandlers(
  message = function(c) cat("Caught a message!\n"),
  {
    message("Someone there?")
    message("Why, yes!")
  }
)
#> Caught a message!
#> Someone there?
#> Caught a message!
#> Why, yes!
```

Handlers are applied in order, so you don't need to worry getting caught in an infinite loop:

```
withCallingHandlers(
  message = function(cnd) message("Second message"),
  message("First message")
)
#> Second message
#> First message
```

(But beware if you have multiple handlers, and some handlers signal conditions that could be captured by another handler: you'll need to think through the order carefully.)

The return value of an calling handler is ignored because the code continues to execute after the handler completes; where would the return value go? That means that calling handlers are only useful for their side-effects.

One important side-effect unique to calling handlers is the ability to **muffle** the signal. By default, a condition will continue to propogate to parent handlers, all the way up to the default handler (or an exiting handler, if provided):

```
# Bubbles all the way up to default handler which generates the message
withCallingHandlers(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
```

```
#> Level 1
#> Level 2
#> Hello

# Bubbles up to tryCatch
tryCatch(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
#> Level 1
#> Level 2
```

If you want to prevent the condition "bubbling up" but still run the rest of the code in the block, you need to explicitly muffle it with `rlang::cnd_muffle()`:

```
# Muffles the default handler which prints the messages
withCallingHandlers(
  message = function(cnd) {
    cat("Level 2\n")
    cnd_muffle(cnd)
  },
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
#> Level 1
#> Level 2

# Muffles level 2 handler and the default handler
withCallingHandlers(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) {
      cat("Level 1\n")
      cnd_muffle(cnd)
    },
    message("Hello")
  )
```

```
)
#> Level 1
```

7.4.4 Call stacks

To complete the section, there are some important differences between the call stacks of exiting and calling handlers. These differences are generally not important but I'm including it here because I've occassionally found it useful, and don't want to forget about it!

It's easiest to see the difference by setting up a small example that uses `lobstr::cst()`:

```
f <- function() g()
g <- function() h()
h <- function() message("!")
```

Calling handlers are called in the context of the call that signalled the condition:

```
withCallingHandlers(f(), message = function(cnd) {
  lobstr::cst()
  cnd_muffle(cnd)
})
#> x
#> +-withCallingHandlers(...)
#> +-f()
#> | \-g()
#> |   \-h()
#> |     \-message("!")
#> |       +-withRestarts(...)
#> |       | \-withOneRestart(expr, restarts[[1L]])
#> |       |   \-doWithOneRestart(return(expr), restart)
#> |       \-signalCondition(cond)
#> \-(function (cnd) ...
#>   \-lobstr::cst()
```

Whereas exiting handlers are called in the context of the call to `tryCatch()`:

```
tryCatch(f(), message = function(cnd) lobstr::cst())
#> x
```

```
#> \-tryCatch(f(), message = function(cnd) lobstr::cst())
#>   \-tryCatchList(expr, classes, parentenv, handlers)
#>     \-tryCatchOne(expr, names, parentenv, handlers[[1L]])
#>       \-value[[3L]](cond)
#>         \-lobstr::cst()
```

7.4.5 Exercises

1. Predict the results of evaluating the following code

```
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      NULL
    }
  )
}

show_condition(stop("!"))
show_condition(10)
show_condition(warning("?!"))
show_condition({
  10
  message("?")
  warning("?!")
})
```

2. Explain the results of running this code:

```
withCallingHandlers(
  message = function(cnd) message("b"),
  withCallingHandlers(
    message = function(cnd) message("a"),
    message("c")
  )
)
```

```
#> b
#> a
#> b
#> c
```

3. Read the source code for `catch_cnd()` and explain how it works.
4. How could you rewrite `show_condition()` to use a single handler?

7.5 Custom conditions

One of the challenges of error handling in R is that most functions generate one of the built-in conditions, which contain only a `message` and a `call`. That means that if you want to detect a specific type of error, you can only work with the text of the error message. This is error prone, not only because the message might change over time, but also because messages can be translated into other languages.

Fortunately R has a powerful, but little used feature: the ability to create custom conditions that can contain additional metadata. Creating custom conditions is a little fiddly in base R, but `rlang::abort()` makes it very easy as you can supply a custom `.subclass` and additional metadata.

The following example shows the basic pattern. I recommend using the following call structure for custom conditions. This takes advantage of R's flexible argument matching so that the name of the "type" of error comes first, followed by the user facing text, followed by custom metadata.

```
abort(
  "error_not_found",
  message = "Path `blah.csv` not found",
  path = "blah.csv"
)
#> Error: Path `blah.csv` not found
```

Custom conditions work just like regular conditions when used interactively, but allow handlers to do much more.

7.5.1 Motivation

To explore these ideas in more depth, let's take `base::log()`. It does the minimum when throwing errors caused by invalid arguments:

```
log(letters)
#> Error in log(letters):
#>   non-numeric argument to mathematical function
log(1:10, base = letters)
#> Error in log(1:10, base = letters):
#>   non-numeric argument to mathematical function
```

I think we can do better by being explicit about which argument is the problem (i.e. `x` or `base`), and saying what the problematic input is (not just what it isn't).

```
my_log <- function(x, base = exp(1)) {
  if (!is.numeric(x)) {
    abort(paste0("`x` must be a numeric vector; not ", typeof(x), "."))
  }
  if (!is.numeric(base)) {
    abort(paste0("`base` must be a numeric vector; not ", typeof(base), "."))
  }

  base::log(x, base = base)
}
```

This gives us:

```
my_log(letters)
#> Error: `x` must be a numeric vector; not character.
my_log(1:10, base = letters)
#> Error: `base` must be a numeric vector; not character.
```

This is an improvement for interactive usage as the error messages are more likely to guide the user towards a correct fix. However, they're no better if you want to programmatically handle the errors: all the useful metadata about the error is jammed into a single string.

7.5.2 Signalling

Let's build some infrastructure to improve this situtation, We'll start by providing a custom `abort()` function for bad arguments. This is a little over-

generalised for the example at hand, but it reflects common patterns that I've seen across other functions. The pattern is fairly simple. We create a nice error message for the user, using `glue::glue()`, and store metadata in the condition call for the developer.

```
abort_bad_argument <- function(arg, must, not = NULL) {
  msg <- glue::glue("`{arg}` must {must}")
  if (!is.null(not)) {
    not <- typeof(not)
    msg <- glue::glue("{msg}; not {not}.")
  }

  abort("error_bad_argument",
    message = msg,
    arg = arg,
    must = must,
    not = not
  )
}
```

In Base R

If you want to throw a custom error without adding a dependency on rlang, you can create a condition object "by hand" and then pass it to `stop()`:

```
stop_custom <- function(.subclass, message, call = NULL, ...) {
  err <- structure(
    list(
      message = message,
      call = call,
      ...
    ),
    class = c(.subclass, "error", "condition")
  )
  stop(err)
}

err <- catch_cnd(stop_custom("error_new", "This is a custom error", x = 10))
class(err)
err$x
```

We can now rewrite `my_log()` to use this new helper:

```
my_log <- function(x, base = exp(1)) {
  if (!is.numeric(x)) {
    abort_bad_argument("x", must = "be numeric", not = x)
  }
  if (!is.numeric(base)) {
    abort_bad_argument("base", must = "be numeric", not = base)
  }

  base::log(x, base = base)
}
```

`my_log()` itself is not much shorter, but is a little more meaninful, and it ensures that error messages for bad arguments are consistent across functions. It yields the same interactive error messages as before:

```
my_log(letters)
#> Error: `x` must be numeric; not character.
my_log(1:10, base = letters)
#> Error: `base` must be numeric; not character.
```

7.5.3 Handling

These structured condition objects are much easier to program with. The first place you might want to use this capability is when testing your function. Unit testing is not a subject of this book (see R packages (`http://r-pkgs.had.co.nz/`) for details), but the basics are easy to understand. The following code captures the error, and then asserts it has the structure that we expect.

```
library(testthat)
#>
#> Attaching package: 'testthat'
#> The following objects are masked from 'package:rlang':
#>
#>     is_false, is_null, is_true

err <- catch_cnd(my_log("a"))
expect_s3_class(err, "error_bad_argument")
expect_equal(err$arg, "x")
expect_equal(err$not, "character")
```

We can also use the class (`error_bad_argument`) in `tryCatch()` to only handle that specfic error:

```
tryCatch(
  error_bad_argument = function(cnd) "bad_argument",
  error = function(cnd) "other error",
  my_log("a")
)
#> [1] "bad_argument"
```

Note that when using `tryCatch()` with multiple handlers and custom classes, the first handler to match any class in the signal's class vector is called, not the best match. For this reason, you need to make sure to put the most specific handlers first. The following code does not do what you might hope:

```
tryCatch(
  error = function(cnd) "other error",
  error_bad_argument = function(cnd) "bad_argument",
  my_log("a")
)
#> [1] "other error"
```

7.5.4 Exercises

1. Inside a package, it's occassionally useful to check that a package is installed before using it. Write a function that checks if a package is installed (with `requireNamespace("pkg", quietly = FALSE)`) and if not, throws a custom condition that includes the package name in the metadata.

2. Inside a package you often need to stop with an error when something is not right. Other packages that depend on your package might be tempted to check these errors in their unit tests. How could you help these packages to avoid relying on the error message which is part of the user interface rather than the API and might change without notice?

7.6 Applications

Now that you've learned the basic tools of R's condition system, it's time to dive into some applications. The goal of this section is not to show every possible usage of `tryCatch()` and `withCallingHandlers()` but to illustrate some

common patterns that frequently crop up. Hopefully these will get your creative juices flowing, so when you encounter a new problem you can come up with a useful solution.

7.6.1 Failure value

There are a few simple, but useful, `tryCatch()` patterns based on returning a value from the error handler. The simplest case is a wrapper to return a "default" value if an error occurs:

```
fail_with <- function(expr, value = NULL) {
  tryCatch(
    error = function(cnd) value,
    expr
  )
}

fail_with(log(10), NA_real_)
#> [1] 2.3
fail_with(log("x"), NA_real_)
#> [1] NA
```

A more sophisticated application is `base::try()`. Below, `try2()` extracts the essense of `base::try()`; the real function is more complicated in order to make the error message look more like what you'd see if `tryCatch()` wasn't used.

```
try2 <- function(expr, silent = FALSE) {
  tryCatch(
    error = function(cnd) {
      msg <- conditionMessage(cnd)
      if (!silent) {
        message("Error: ", msg)
      }
      structure(msg, class = "try-error")
    },
    expr
  )
}

try2(1)
#> [1] 1
try2(stop("Hi"))
#> Error: Hi
```

```
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
try2(stop("Hi"), silent = TRUE)
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
```

7.6.2 Success and failure values

We can extend this pattern to returns one value if the code evaluates successfully (`success_val`), and another if it fails (`error_val`). This pattern just requires one small trick: evaluating the user supplied code, then `success_val`. If the code throws an error, we'll never get to `success_val` and will instead return `error_val`.

```
foo <- function(expr) {
  tryCatch(
    error = function(cnd) error_val,
    {
      expr
      success_val
    }
  )
}
```

We can use this to determine if an expression fails:

```
does_error <- function(expr) {
  tryCatch(
    error = function(cnd) TRUE,
    {
      expr
      FALSE
    }
  )
}
```

Or to capture any condition, like just `rlang::catch_cnd()`:

```
catch_cnd <- function(expr) {
  tryCatch(
    condition = function(cnd) cnd,
    {
      expr
      NULL
    }
  )
}
```

We can also use this pattern to create a `try()` variant. One challenge with `try()` is that it's slightly challenging to determine if the code succeeded or failed. Rather than returning an object with a special class, I think it's slightly nicer to return a list with two components `result` and `error`.

```
safety <- function(expr) {
  tryCatch(
    error = function(cnd) {
      list(result = NULL, error = c)
    },
    list(result = expr, error = NULL)
  )
}

str(safety(1 + 10))
#> List of 2
#>  $ result: num 11
#>  $ error : NULL
str(safety(abort("Error!")))
#> List of 2
#>  $ result: NULL
#>  $ error :function (...)
```

(This is closely related to `purrr::safely()`, an adverb.)

7.6.3 Resignal

As well as returning default values when a condition is signalled, handlers can be used to make more informative error messages. One simple application is to make a function that works like `option(warn = 2)` for a single block of code. The idea is simple: we handle warnings by throwing an error:

```
warning2error <- function(expr) {
  withCallingHandlers(
    warning = function(cnd) abort(conditionMessage(cnd)),
    expr
  )
}
```

```
warning2error({
  x <- 2 ^ 4
  warn("Hello")
})
#> Error: Hello
```

You could write a similar function if you were trying to find the source of an annyoing message.

7.6.4 Record

Another common pattern is to record conditions for later investigation. The new challenge here is that calling handlers are called only for their side-effects so we can't return values, but instead need to modify some object in place.

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
    cnd_muffle(cnd)
  }

  withCallingHandlers(
    message = add_cond,
    warning = add_cond,
    expr
  )

  conds
}

catch_cnds({
  inform("a")
  warn("b")
```

```
  inform("c")
})
#> [[1]]
#> <message: a
#> >
#>
#> [[2]]
#> <warning: b>
#>
#> [[3]]
#> <message: c
#> >
```

What if you also want to capture errors? You'll need to wrap the `withCallingHandlers()` in a `tryCatch()`. If an error occurs, it will be the last condition.

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
    cnd_muffle(cnd)
  }

  tryCatch(
    error = function(cnd) {
      conds <<- append(conds, list(cnd))
    },
    withCallingHandlers(
      message = add_cond,
      warning = add_cond,
      expr
    )
  )

  conds
}

catch_cnds({
  inform("a")
  warn("b")
  abort("C")
})
#> [[1]]
```

```
#> <message: a
#> >
#>
#> [[2]]
#> <warning: b>
#>
#> [[3]]
#> <rlang_error: C>
```

This is the key idea underlying the evaluate (`https://github.com/r-lib/evaluate`) package which powers knitr: it captures every output into a special data structure so that it can be later replayed. As a whole, the evaluate package is quite a lot more complicated than the code here because it also needs to handle plots and text output.

7.6.5 No default behaviour

A final useful pattern is to signal a condition that doesn't inherit from `message`, `warning` or `error`. Because there is no default behaviour, this means the condition has no effect unless the user specifically requests it. For example, you could imagine a logging system based on conditions:

```
log <- function(message, level = c("info", "error", "fatal")) {
  level <- match.arg(level)
  signal(message, "log", level = level)
}
```

When you call `log()` a condition is signalled, but nothing happens because it has no default handler:

```
log("This code was run")
```

To "activate" logging you need a handler that does something with the `log` condition. Below I define a `record_log()` function that will record all logging messages to a path:

```
record_log <- function(expr, path = stdout()) {
  withCallingHandlers(
    log = function(cnd) {
      cat(
        "[", cnd$level, "] ", cnd$message, "\n", sep = "",
```

```
        file = path, append = TRUE
      )
    },
    expr
  )
}

record_log(log("Hello"))
#> [info] Hello
```

You could even imagine layering with another function that allows you to selectively suppress some logging levels.

```
ignore_log_levels <- function(expr, levels) {
  withCallingHandlers(
    log = function(cnd) {
      if (cnd$level %in% levels) {
        cnd_muffle(cnd)
      }
    },
    expr
  )
}

record_log(ignore_log_levels(log("Hello"), "info"))
```

In Base R

If you create a condition object by hand, and signal it with `signalCondition()`, `cnd_muffle()` will not work. Instead you need to call it with a muffle restart defined, like this:

```
withRestarts(signalCondition(cond), muffle = function() NULL)
```

Restarts are currently beyond the scope of the book, but I suspect will be included in the 3rd edition.

7.6.6 Exercises

1. Create `suppressConditions()` that works like `suppressMessages()` and `supressWarnings()` but supresses everything. Think carefully about how you should handle errors.

2. 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))
}
```

3. How would you modify the `catch_cnds()` defined if you wanted to recreate the original intermingling of warnings and messages?

4. Why is catching interrupts dangerous? Run this code to find out.

```
bottles_of_beer <- function(i = 99) {
  message("There are ", i, " bottles of beer on the wall, ", i, " bottles of beer.")
  while(i > 0) {
    tryCatch(
      Sys.sleep(1),
      interrupt = function(err) {
        i <<- i - 1
        if (i > 0) {
          message(
            "Take one down, pass it around, ", i,
            " bottle", if (i > 1) "s", " of beer on the wall."
          )
        }
      }
    )
  }
  message("No more bottles of beer on the wall, no more bottles of beer.")
}
```

7.7 Quiz answers

1. `error`, `warning`, and `message`.
2. You could use `try()` or `tryCatch()`.
3. `tryCatch()` creates exiting handlers which will terminate the execution of wrapped code; `withCallingHandlers()` creates calling handlers which don't affect the execution of wrapped code.
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.

8

Connections

In R, every time you read data in or write data out, you are using a connection behind the scenes. Connections abstract away the underlying implementation so that you can read and write data the same way, regardless of whether you're writing to a file, an HTTP connnection, a pipe, or something more exotic.

- `http://biostatmatt.com/R/R-conn-ints/index.html#Top`
- `?file`
- `https://cran.r-project.org/doc/Rnews/Rnews_2001-1.pdf`
- `https://cran.r-project.org/doc/manuals/r-release/R-data.html#Connections`

8.1 Basics

- default connections: stdin, stderr, stdout
- `cat()` + `cat_line()`
- survey of base connections: file, compressed file, url, pipe, socket, text
- important packages: curl
- blocking vs non-blocking
- pattern: `close()` with `on.exit()` if you opened

8.2 Reading and writing binary data

- `raw()`
- `readBin()` vs `writeBin()`
- text vs binary (newlines and nulls)

8.3 Reading and writing text data

Reading and writing text is more complicated than reading and writing binary data because as soon as you move beyond regular ASCII characters (e.g. a-z, 0-9) there are many different ways of representing the same text. The way in which text data is stored in binary is known as the **encoding**.

- Encodings
  - `https://kevinushey.github.io/blog/2018/02/21/string-encoding-and-r/`
  - in general vs `Encoding`
  - `encoding` vs `fileEncoding`
  - converting with iconv
- UTF-8 everywhere
- Reliably reading and writing UTF-8

Part II

Functional programming

Introduction

R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions.

FP is a technical term that defines certain properties of a programming language. But it's also a programming style.

In R, the tools for working with functions, the tools of functional programming, are more important than the tools for working with objects or classes, the tools of object oriented programming.

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. A higher-order function is a function that takes a function as an input or returns a function as output. This part of the book is broken down by the three types of higher-order functions:

- **Functionals**, Chapter **??** where functions are the input. Functionals are by far and away the most immediately useful application of FP ideas, and you'll use them all the time in data analyses.
- **Function factories**, Chapter **??**, functions as output. You can almost always avoid function factories in favour of a different technique, but they are occassionally useful.
- **Function operators**, Chapter **??**, discusses function as input and output. These are like adverbs, because they typically modify the operation of a function.

8.3.0.1 Other languages

While FP techniques form the core of languages like Haskell, OCaml and F#, those techniques can also be found in other languages. They are well supported in multi-paradigm systems like Lisp, Scheme, Clojure and Scala. Also, while they tend not to be the dominant technique used by programmers, they can be used in modern scripting languages like Python, Ruby and JavaScript. In contrast, C, Java and C# provide few functional tools, and while it's possible to do FP in those languages, it tends to be an awkward fit. In sum, if you

Google for it you're likely to find a tutorial on functional programming in any language. But doing so can often be syntactically awkward or used so rarely that other programmers won't understand your code.

Recently FP has experienced a surge in interest because it provides a complementary set of techniques to object oriented programming, the dominant style for the last several decades. Since FP functions tend to not modify their inputs, they make for programs that are easier to reason about using only local information, and are often easier to parallelise. The traditional weaknesses of FP languages, poorer performance and sometimes unpredictable memory usage, have been largely eliminated in recent years.

9

Functionals

9.1 Introduction

> "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 **functional** is 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. You might have used for-loop replacement like base R's `lapply()`, `apply()`, or `tapply()`, or maybe purrr's `map()` or variant; or maybe you've used a mathemetical functional like `integrate()` or `optim()`. All functionals 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 Section **??** 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 immediately know 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 and purrr 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 Section **??**.

Using functionals is a pattern matching exercise. You look at the for loop, and find a functional that matches the basic form. If one doesn't exist, don't try and torture an existing functional to fit the form you need. Instead, just leave it as a for loop!

It's not about eliminating for loops. It's about having someone else write them for you!

Outline

Prerequisites

This chapter will focus on functionals provided by the purrr package. These functions have a consistent interface that makes it easier to understand the key ideas than their base equivalents, which have grown organically over many years. I'll compare and contrast base R functions as we go, and then wrap up the chapter with a discussion of base functionals that don't have purrr equivalents.

```
library(purrr)
```

Many R users feel guilty about using for loops instead of apply functions. It's natural to blame yourself for failing to understand and internalise the apply family of functions. However, I think this is like blaming yourself when embarass yourself by failing to pull open a door when it's supposed to be pushed open[1]. It's not actually your fault, because many people suffer the same problem; it's a failing of design. Similarly, I think the reason why the apply functions are so hard for so many people is because their design is suboptimal.

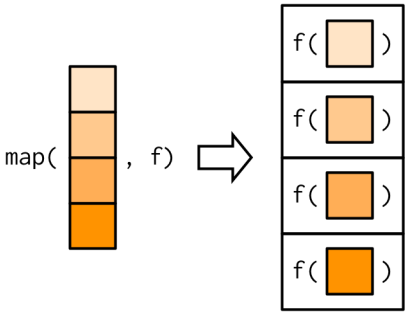
[1] These are sometimes called Norman doors after Don Norman who described them in his book, "The Design of Everyday Things". There's a nice video about them at https://99percentinvisible.org/article/norman-doors/.

9.2 My first functional: map()

The most fundamental functional is purrr::map()[2]. It takes a vector and a function, calls the function once for each element of the vector, and returns the results in a list. In other words, map(1:3, f) yields list(f(x[[1]]), f(x[[2]]), f(x[[3]])).

```
triple <- function(x) x * 3
map(1:3, triple)
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 6
#>
#> [[3]]
#> [1] 9
```

Or, graphically:



You might wonder why this function is called map(). What does it have to do with depicting physical features of land or sea ? In fact, the meaning comes from mathematics where map refers to "an operation that associates each element of a given set with one or more elements of a second set". This makes sense here because map() defines a mapping from one vector to another. ("Map" also has the nice property of being short, which useful for such a fundamental building block.)

[2]Not to be confused with base::Map(), which is considerably more complex, and we'll come back to in Section **??**.

The implementation of `map()` is quite simple. We allocate a list the same length as the input, and then fill in the list with a for loop. The basic implementation is only a handful of lines of code:

```
simple_map <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

The real `purrr::map()` function has a few differences: it is written in C to eke out every last iota of performance, preserves names, and supports a few shortcuts that you'll learn about shortly.

In Base R

The base equivalent to `map()` is `lapply()`. The only difference is that `lappy()` does not support the helpers that you'll learn about below, so if you're only using `map()` from purrr, you can skip the additional package and use `base::lapply()` directly.

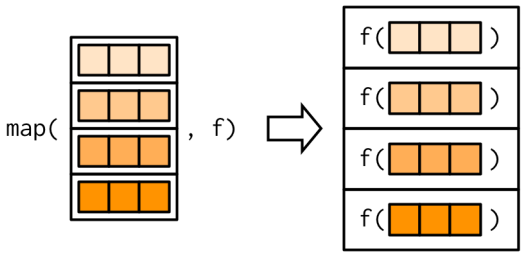
9.2.1 Producing atomic vectors

`map()` returns a list. This makes `map()` the most general of the "map" family because you can put anything in a list. There are four more specific variants, `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()`, that return atomic vectors:

```
map_chr(mtcars, typeof)
#>      mpg      cyl     disp       hp     drat       wt     qsec
#> "double" "double" "double" "double" "double" "double" "double"
#>       vs       am     gear     carb
#> "double" "double" "double" "double"

map_lgl(mtcars, is.double)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

map_dbl(mtcars, mean)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs
#>  20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438
```

```
#>      am    gear    carb
#>   0.406   3.688   2.812

n_unique <- function(x) length(unique(x))
map_int(mtcars, n_unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

These examples rely on the fact that data frames are lists containing vectors of the same length:



Like map(), the input and the output must be the same length, so you can not return multiple values. When debugging problems like this, it's often useful to switch back to map() so you can see what the problematic output is.

```
pair <- function(x) c(x, x)
map_dbl(1:3, pair)
#> Error: Result 1 is not a length 1 atomic vector

map(1:3, pair)
#> [[1]]
#> [1] 1 1
#>
#> [[2]]
#> [1] 2 2
#>
#> [[3]]
#> [1] 3 3
```

```
simple_map_dbl <- function(x, f, ...) {
  out <- double(length(x))
  for (i in seq_along(x)) {
    val <- f(x[[i]], ...)
    if (length(val) != 1 || !is.numeric(out)) {
```

```
      stop("Result ", i, " is not a length 1 atomic vector", call. = FALSE)
    }
    out[[i]] <- val
  }
  out
}
```

In Base R

Base R has two similar functions: `sapply()` and `vapply()`.

`sapply()` tries to simplify the result to an atomic vector, whereever possible. But this simplifiation depends on the input, so sometimes you'll get a list, sometimes a vector, and sometimes a matrix. This makes it difficult to program with.

`vapply()` allows you to provide a template that describes the output shape. If you want to stick to with base R code you should always use `vapply()` in your functions, not `sapply()`. The primary downside of `vapply()` is its vebosity: the equivalent to `map_dbl(x, mean, na.rm = TRUE)` is `vapply(x, mean, na.rm = TRUE, FUN.VALUE = double())`.

9.2.2 Anonymous functions and helpers

Instead of using `map()` with an existing function, you can create an inline anonymous function (as mentioned in Section **??**first-class-functions)):

```
map_dbl(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
```

Anonymous functions are very useful, but the syntax is verbose. So purrr offers a shorthand:

```
map_dbl(mtcars, ~ 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
```

That also makes for a handy way of generating random data:

```
x <- map(1:3, ~ runif(2))
str(x)
```

```
#> List of 3
#>  $ : num [1:2] 0.281 0.53
#>  $ : num [1:2] 0.433 0.917
#>  $ : num [1:2] 0.0275 0.8249
```

Reserve this syntax for short and simple functions. A good rule of thumb is that if your function involves spans lines or uses `{}`, it's time to name your function.

Inside all purrr functions you can create an anonymous function using a ~ (the usual formula operator, pronounced "twiddle"). You can see what happens by calling `as_mapper()`: the map functions normally do that for you, but it's useful to do it "by hand" to see what's going on:

```
as_mapper(~ length(unique(.x)))
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> length(unique(.x))
```

The function arguments look a little quirky but allow you to refer to `.` for one argument functions, `.x` and `.y` for two argument functions, and `..1`, `..2`, `..3`, etc, for functions with an arbitrary number of arguments.

purrr also provides helpers for extracting elements from a vector, powered by `purrr::pluck()`. You can use a character vector to select elements by name, an integer vector to select by position, or a list to select by both name and position. These are very useful for working with deeply nested lists, which often arise when working with JSON.

```
x <- list(
  list(-1, x = 1, y = c(2), z = "a"),
  list(-2, x = 4, y = c(5, 6), z = "b"),
  list(-3, x = 8, y = c(9, 10, 11))
)

# Select by name
map_dbl(x, "x")
#> [1] 1 4 8

# Or by position
map_dbl(x, 1)
#> [1] -1 -2 -3

# Or by both
map_dbl(x, list("y", 1))
```

```
#> [1] 2 5 9

# You'll get an error if a component doesn't exist:
map_chr(x, "z")
#> Error: Result 3 is not a length 1 atomic vector
# Unless you supply a .default value
map_chr(x, "z", .default = NA)
#> [1] "a" "b" NA
```

In Base R

In base R functions, like `lapply()`, you can provide the name of the function as a string. This isn't tremendously useful as most of the time `lapply(x, "f")` is exactly equivalent to `lapply(x, f)`, just more typing.

9.2.3 Passing arguments with ...

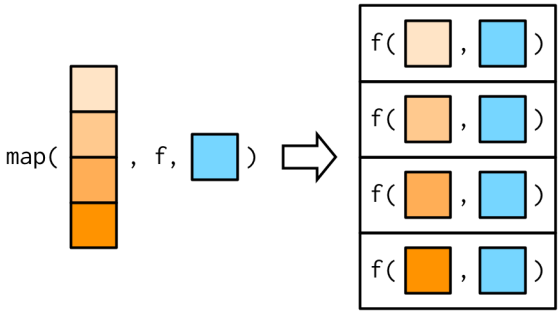
It's often convenient to pass on along additional arguments to the function that you're calling. For example, you might want to pass `na.rm = TRUE` along to `mean()`. One way to do that is with an anonymous function:

```
x <- list(1:5, c(1:10, NA))
map_dbl(x, ~ mean(.x, na.rm = TRUE))
#> [1] 3.0 5.5
```

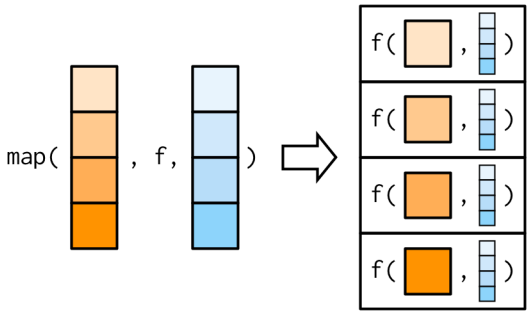
But because the map functions pass ... along, there's a simpler form available:

```
map_dbl(x, mean, na.rm = TRUE)
#> [1] 3.0 5.5
```

This is easiest to understand with a picture: any arguments that come after `f` in the call to `map()` are inserted *after* the data in individual calls to `f()`:



It's important to note that these arguments are not decomposed; or said another way, `map()` is only vectorised over its first argument. If an argument after `f` is a vector, it will be passed along as is, not decomposed like the first argument:



Note there's a subtle difference between placing extra arguments inside an anonymous function compared with passing them to `map()`. Putting them in anonymous function means that they will be evaluated every time `f()` is executed, not just once when you call `map()`. This is easiest to see if we make the additional argument random:

```
plus <- function(x, y) x + y

x <- c(0, 0, 0, 0)
map_dbl(x, plus, runif(1))
#> [1] 0.0625 0.0625 0.0625 0.0625
map_dbl(x, ~ plus(.x, runif(1)))
#> [1] 0.903 0.132 0.629 0.945
```

9.2.4 Argument names

In the diagrams, I've omitted argument names to focus on the overall structure. But I recommend writing out the full names in your code, as it makes it easier to read. `map(x, mean, 0.1)` is perfectly valid code, but it generates `mean(x[[1]], 0.1)` so it relies on the reader remembering that the second argument to `mean()` is `trim`. To avoid unnecesary burden on the brain of the reader[3], be kind, and write `map(x, mean, trim = 0.1)`.

This is the reason why the arguments to `map()` are a little odd: instead of being `x` and `f`, they are `.x` and `.f`. It's easiest to the problem that leads to these names using `simple_map()` defined above. `simple_map()` has arguments `x` and `f` so you'll have problems whenever the function you are calling has arguments `x` or `f`:

```
boostrap_summary <- function(x, f) {
  f(sample(x, replace = TRUE))
}

simple_map(mtcars, boostrap_summary, f = mean)
#> Error in mean.default(x[[i]], ...):
#>   'trim' must be numeric of length one
```

The error is a little bewildering until you remember that the call to `simple_map()` is equivalent to `simple_map(x = mtcars, f = mean, bootstrap_summary)` because named matching beats positional matching.

purrr functions reduce the likelihood of such a clash by using `.f` and `.x` instead of the more common `f` and `x`. Of course this technique isn't perfect (because the function you are calling might still use `.f` and `.x`), but it avoids 99% of issues. The remaining 1% of the time, use an anonymous function.

In Base R

Base functions that pass along `...` use a variety of naming conventions to prevent undesired argument matching:

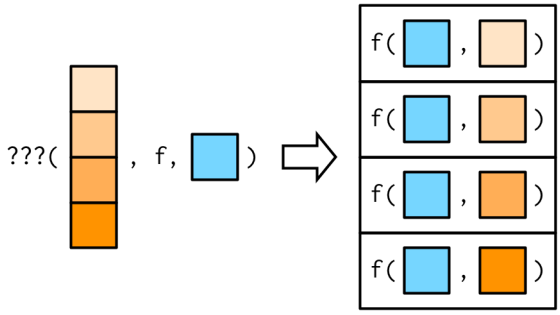
- The apply family mostly uses capital letters (e.g `X` and `FUN`).
- `transform()` uses more exotic prefix `_`: this makes the name non-syntactic so it must always be surrounded in `` ` ``, as described in Section **??**. This makes undesired matches extremely unlikely.
- Other functional like `uniroot()` and `optim()` make no effort to avoid clashes;

[3] Who is highly likely to be future you!

but they tend to be used with specially created funtions so clashes are less likely.

9.2.5 Varying another argument

So far the first argument to `map()` has always become the first argument to the function. But what happens if the first argument should be constant, and you want to vary a different argument? How do you get the result in this picture?



It turns out that there's no way to do it directly, but there are two tricks you can use. To illustrate them, imagine I have a vector that contains a few unusual values, and I want to explore the effective of different amounts of trimming when computing the mean. In this case, the first argument to `mean()` will be constant, and I want to vary the second argument, `trim`.

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)
```

- The simplest technique is to use an anonymous function to rearrange the argument order:

```
map_dbl(trims, ~ mean(x, trim = .x))
#> [1] -0.3500  0.0434  0.0354  0.0502
```

- Sometimes, if you want to be (too) clever, you can take advantage of R's flexible argument matching rules (as described in Section **??**). For example, in this example you can rewrite `mean(x, trim = 0.1)` as `mean(0.1, x = x)`, so you could write the call to `map_dbl()` as:

```
map_dbl(trims, mean, x = x)
#> [1] -0.3500  0.0434  0.0354  0.0502
```

I don't recommend this technique as it relies on the reader being very familiar with both the argument order to `.f`, and R's argument matching rules.

9.2.6 Exercises

1. Use `as_mapper()` to explore how purrr generates anonymous functions for the integer, character, and list helpers. What helper allows you to extract attributes? Read the documentation to find out.

2. `map(1:3, ~ runif(2))` is a useful pattern for generating random numbers, but `map(1:3, runif(2))` is not. Why not? Can you explain why it returns the result that it does?

3. Use the appropriate `map()` function 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 do it in two steps.)

   c) Compute the number of levels for in every factor in a data frame.

4. The following code simulates the performance of a t-test for non-normal data. Extract the p-value from each test, then visualise.

```
trials <- map(1:100, ~ t.test(rpois(10, 10), rpois(7, 10)))
```

5. The following code uses a map nested inside another map to apply a function to every element of a nested list. Why does it fail, and what do you need to do to make it work?

```
x <- list(
  list(1, c(3, 9)),
  list(c(3, 6), 7, c(4, 7, 6))
)

triple <- function(x) x * 3
map(x, map, .f = triple)
#> Error in .f(.x[[i]], ...):
#>   unused argument (map)
```

6. Use `map()` 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
)
```

7. Fit the model `mpg ~ disp` to each of the bootstrap replicates of `mtcars` in the list below, then extract the R^2 of the model fit (Hint: you can compute the R^2 with `summary()`)

```
bootstrap <- function(df) {
  df[sample(nrow(df), replace = TRUE), , drop = FALSE]
}

bootstraps <- map(1:10, ~ bootstrap(mtcars))
```

9.3 Map variants

There are 23 primary variants of `map()`. So far, you've learned about five (`map()`, `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()`). This section will introduce you to 18 (!!) more. That sounds like a lot but fortunately you only need to learn five new ideas:

- Output same type as input with `modify()`
- Iterate over two inputs with `map2()`.
- Iterate with an index using `imap()`
- Return nothing with `walk()`.
- Iterate over any number of inputs with `pmap()`.

The design of purrr makes the input and output orthogonal, forming the rows and columns of the matrix below. Once you've mastered the idea in a row, you can combine it with any column; once you've mastered the idea in column, you can combine it with any row. This orthogonality makes purrr easier to learn.

	List	Atomic	Same type	Nothing	
One argument		`map()`	`map_lgl()`, …	`modify()`	`walk()`
Two arguments		`map2()`	`map2_lgl()`, …	`modify2()`	`walk2()`
One argument + index		`imap()`	`imap_lgl()`, …	`imodify()`	`iwalk()`
N arguments		`pmap()`	`pmap_lgl()`, …	—	`pwalk()`

9.3.1 Same type of output as input: `modify()`

Imagine you wanted to double every column in a data frame. You might first try using `map()`, but `map()` always returns a list:

```
df <- data.frame(
  x = 1:3,
  y = 6:4
)

map(df, ~ .x * 2)
#> $x
#> [1] 2 4 6
#>
#> $y
#> [1] 12 10  8
```

If you want to keep the output as a data frame, you can use `modify()`, which always returns the same type of output as the input:

```
modify(df, ~ .x * 2)
#>   x  y
#> 1 2 12
#> 2 4 10
#> 3 6  8
```

Of course, `modify()` doesn't modify in place, it returns a modified copy, so if you wanted to permanently modify `df`, you'd need to assign it:

```
df <- modify(df, ~ .x * 2)
```

As usual, the basic implementation of `modify()` is simple. It's even simpler than `map()` because we don't need to create a new output vector; we can just progressively replace the input. The real code is a little complex to handle edge cases more gracefully.

```
simple_modify <- function(x, f, ...) {
  for (i in seq_along(x)) {
    x[[i]] <- f(x[[i]], ...)
  }
  x
}
```

In Section @(predicate-map) you'll learn about a very useful variant of `modify()`, called `modify_if()`. This allows you to (e.g.) only double numeric columns of a data frame with `modify(df, is.numeric, ~ .x * 2)`.

9.3.2 Two inputs: `map2()` and friends

`map()` is vectorised over a single argument, i.e. it only varies `.x` when calling `.f`. This makes it poorly suited for some problems. For example, how would you find a weighted mean when you have a list of observations and a list of weights? Imagine we have the following data:

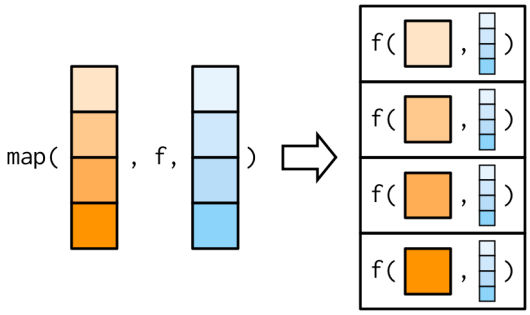
```
xs <- map(1:8, ~ runif(10))
xs[[1]][[1]] <- NA
ws <- map(1:8, ~ rpois(10, 5) + 1)
```

You can use `map_dbl()` to compute the unweighted means:

```
map_dbl(xs, mean)
#> [1]    NA 0.463 0.551 0.453 0.564 0.501 0.371 0.443
```

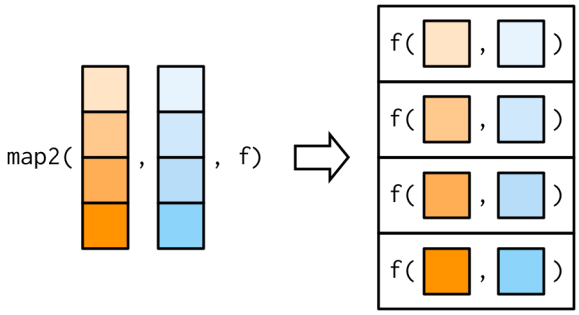
But passing `ws` as an additional argument doesn't work because arguments after `.f` are passed as is.

```
map_dbl(xs, weighted.mean, w = ws)
#> Error in weighted.mean.default(.x[[i]], ...):
#>   'x' and 'w' must have the same length
```



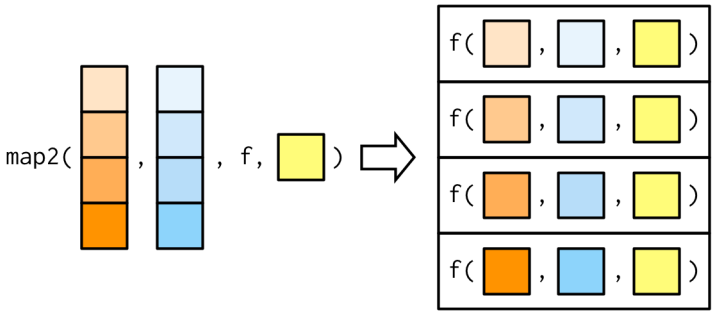
We need a new tool: a `map2()`, which is vectorised over two arguments:

```
map2_dbl(xs, ws, weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



The arguments to `map2()` are slightly different to the arguments to `map()`; two vectors that come before the function instead of one. Additional arguments can still come afterwards:

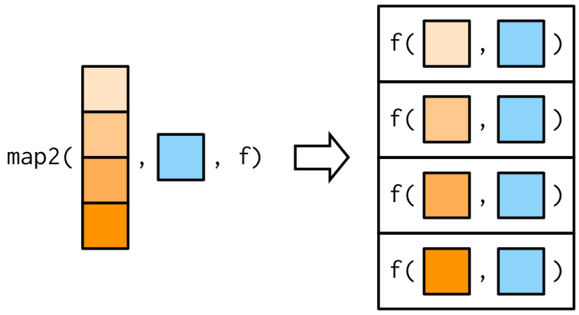
```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



The basic implementation of `map2()` is simple, and quite similar to that of `map()`. Instead of iterating over one vector, we iterate over two in parallel:

```
simple_map2 <- function(x, y, f, ...) {
  out <- vector("list", length(xs))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

One of the big differences between `map2()` and the simple function above is that `map2()` recycles its inputs to make sure that they're the same length:



In other words, `map2(x, y, f)` will automatically behave like `map(x, f, y)` when needed. This is helpful when writing functions; in scripts you'd generally just use the simpler form directly.

In Base R

The closest no base equivalent to `map2()` is `Map()`, which is discussed in Section ??.

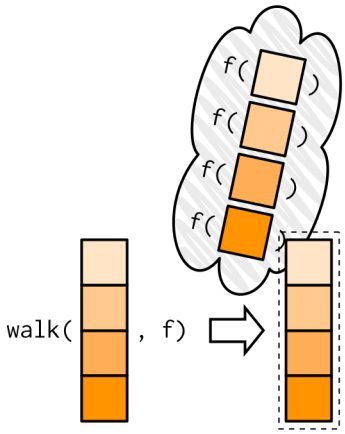
9.3.3 No outputs: `walk()` and friends

Most functions are called for their return value, so it makes sense to capture and store with a `map()` function. But some functions are called primarily for their side-effects, like `cat()`, `write.csv()`, or `ggsave()`, and it doesn't make sense to capture their results. Take this simple example that displays a welcome message using `cat()`. `cat()` returns `NULL`, so while map works (in the sense that it generates the desired welcomes), it also returns `list(NULL, NULL)`.

```
welcome <- function(x) {
  cat("Welcome ", x, "!\n", sep = "")
}
names <- c("Hadley", "Jenny")

# As well as generate the welcomes, it also shows
# the return value of cat()
map(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL
```

You could avoid this problem by assigning the results of `map()` to a variable that you never use, but that would muddy the intent of the code. Instead, purrr provides the walk family of functions that ignore the return values of the `.f` and instead return `.x` invisibly[4].

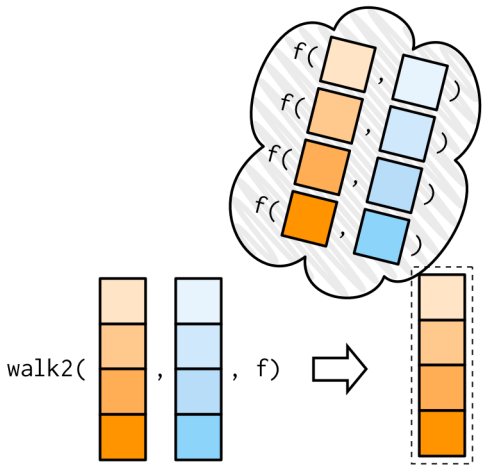
```
walk(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
```



One of the most useful `walk()` variants is `walk2()` because a very common

[4]In brief, invisible values are only printed if you explicitly request them. It's a good idea to return them from functions called primarily for their side-effects because it makes easier to compose, if desired. See Section **??** for more details.

side-effect is saving something to disk, and when saving something to disk you always have a pair of values: the object and the path that you want to save it to.



For example, imagine you have a list of data frames (which I've created here using split), and you'd like to save each one to a separate csv file. That's easy with `walk2()`:

```
temp <- tempfile()
dir.create(temp)

cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(cyls, paths, write.csv)

dir(temp)
#> [1] "cyl-4.csv" "cyl-6.csv" "cyl-8.csv"
```

Here the `walk2()` is equivalent to `write.csv(cyls[[1]], paths[[1]])`, `write.csv(cyls[[2]], paths[[2]])`, `write.csv(cyls[[3]], paths[[3]])`.

In Base R

There is no base equivalent to `walk()`; you can either wrap the result of `lapply()` in `invisible()` or save it to a variable that you never use.

9.3.4 Alternative indices

There are three basic ways to loop over a vector with a for loop:

- Loop over the elements: `for (x in xs)`
- Loop over the numeric indices: `for (i in seq_along(xs))`
- Loop over the names: `for (nm in names(xs))`

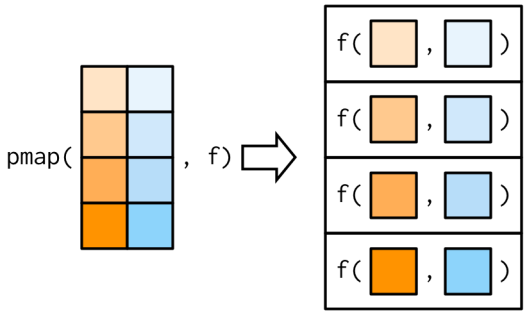
The first form is analogous to the `map()` family. The second and third forms are equivalent to the `imap()` family. `imap()` is like `map2()` in the sense that `.f` gets called with two arguments: `imap(x, f)` is equivalent to `map2(x, names(x), f)` if x has names, and `map2(x, seq_along(x), f)` if it does not.

```
x <- sample(100, 5)
str(imap(x, ~ c(.x, .y)))
#> List of 5
#>  $ : int [1:2] 75 1
#>  $ : int [1:2] 12 2
#>  $ : int [1:2] 30 3
#>  $ : int [1:2] 86 4
#>  $ : int [1:2] 10 5
```

`imap()` is a useful helper if you want to work the values in a vector along with their positions.

9.3.5 Any number of inputs: `pmap()` and friends

Since there's `map()` and `map2()`, you might expect that there's `map3()`, `map4()`, `map5()`, and so on. But where would you stop? Instead of generalisating to an arbitrary number of arguments, purrr takes a slightly different tack with `pmap()`: you supply it with a list of arguments. In most cases, that list will be a list of vectors, where every vector is the same length, which makes it very similar to a data frame. In diagrams, I'll emphasise that structure by putting the inputs side-by-side:

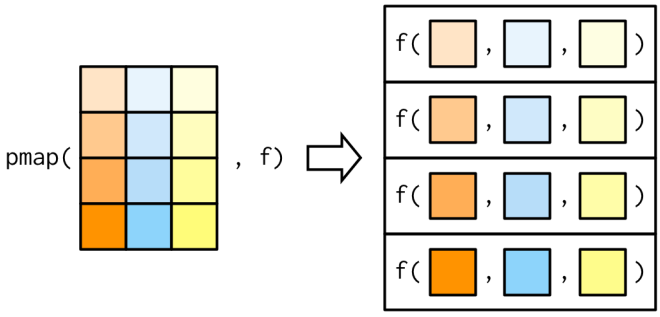


We used `map2_dbl(xs, ws, weighted.mean)` above: the `pmap()` equivalent is:

```
pmap_dbl(list(xs, ws), weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

As before, the varying arguments come before `.f` (although now they must be wrapped in a list), and and the constant arguments come afterwards.

```
pmap_dbl(list(xs, ws), weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



It's often convenient to call `pmap()` with a data frame. A handy way to create that data frame is with `tibble::tribble()`, which allows you to describe a data frame row-by-row (rather than column-by-column, as usual): thinking about the parameters to a function as a data, is a very powerful pattern. The following example shows how you might draw random uniform numbers with varying parameters:

```
params <- tibble::tribble(
  ~ n, ~ min, ~ max,
   1L,     0,     1,
```

```
   2L,    10,   100,
   3L,   100,  1000
)

pmap(params, runif)
#> [[1]]
#> [1] 0.182
#>
#> [[2]]
#> [1] 28.3 15.5
#>
#> [[3]]
#> [1] 903 693 667
```

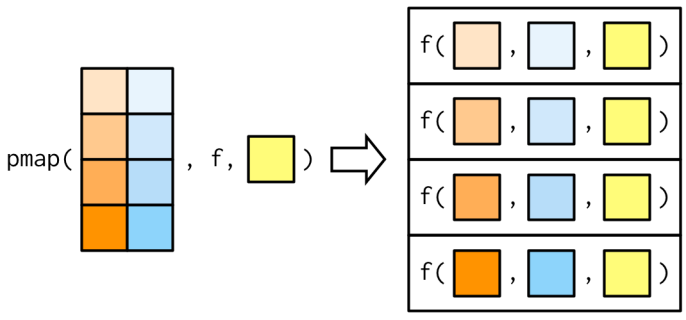
Here, the column names are critical: I've carefully chosen to match them to the arguments to `runif()`, so the `pmap(params, runif)` is equivalent to `runif(n = 1L, min = 0, max = 1)`, `runif(n = 2, min = 10, max = 100)`, `runif(n = 3L, min = 100, max = 1000)`.

In Base R

There are two base equivalents to the `pmap()` family: `Map()` and `mapply()`. Both have significant drawbacks:

- `Map()` vectorises over all arguments so you can not

`Map()` is most similar to `pmap()` but it doesn't provide any way to pass constant arguments. `mapply()` is the multidimensional version of `sapply()`; concetually it takes the output of `Map()` and simplifies it if possible. `mapply()` provides the `MoreArgs` argument which allows you to pass additional constant arguments.



9.3.6 Exercises

1. Explain the results of `modify(mtcars, 1)`.
2. Explain how the following code transforms a data frame using functions stored in a list.

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, labels = c("auto", "manual"))
)

vars <- names(trans)
mtcars[vars] <- map2(trans, mtcars[vars], function(f, var) f(var))
```

   Compare and constrast the `map2()` approach to this `map()` approach:

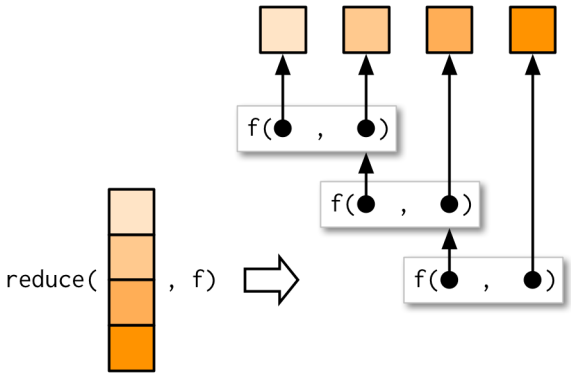
```
mtcars[vars] <- map(vars, ~ trans[[.x]](mtcars[[.x]]))
```

3. What does `write.csv()` return? i.e. what happens if you use it with `map2()` instead of `walk2()`?

9.4 Other important purrr functionals

9.4.1 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(1:3, f)` is equivalent to `f(f(1, 2), 3)`. Reduce is also known as fold, because it folds together adjacent elements in the list.



As usual, the essence of `reduce()` can reduced to a simple wrapper around a for loop:

```
simple_reduce <- function(x, f) {
  out <- x[[1]]
  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

`reduce()` is a useful way to generalise a function that works with two inputs (a **binary** function) to work with any number of inputs. Imagine you have a list of numeric vectors, and you want to find the values that occur in every element:

```
l <- map(1:5, ~ sample(1:10, 15, replace = T))
str(l)
#> List of 5
#>  $ : int [1:15] 9 1 1 8 6 5 2 3 7 10 ...
#>  $ : int [1:15] 6 1 9 1 4 5 10 7 1 10 ...
#>  $ : int [1:15] 7 9 4 6 6 1 9 5 7 2 ...
#>  $ : int [1:15] 4 7 2 7 1 7 2 2 10 7 ...
#>  $ : int [1:15] 4 6 4 10 2 8 6 5 2 8 ...
```

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] 5
```

That's hard to read. With `Reduce()`, the equivalent is:

```
reduce(l, intersect)
#> [1] 5
```

To see how `reduce()` works, it's useful to use a variant: `accumulate()`. As well as returning the final result, `accumulate()` also returns all intermediate results.

```
accumulate(l, intersect)
#> [[1]]
#>  [1]  9  1  1  8  6  5  2  3  7 10  7  6  5  6 10
#>
#> [[2]]
#> [1]  9  1  6  5  3  7 10
#>
#> [[3]]
#> [1] 9 1 6 5 3 7
#>
#> [[4]]
#> [1] 1 5 7
#>
#> [[5]]
#> [1] 5
```

In Base R

The base equivalent is `Reduce()`. It takes the function as the first argument and the vector as second; there is no way to supply additional constant arguments.

9.4.2 Predicate functionals

A **predicate** is a function that returns a single `TRUE` or `FALSE`, like `is.character()`, `is.null()`, or `all()`, and we say a predicate **matches** a vector if it returns `TRUE`. A **predicate functional** applies a predicate to each element of a vector. purrr proivdes six useful functions which come in three pairs:

- `some(.x, .p)` returns `TRUE` if *any* element matches; `every(.x,, .p)` returns `TRUE` if *all* elements match.
- `detect(.x, .p)` returns the *value* of the first match; `detect_index(.x, .p)` returns the *location* of the first match.
- `keep(.x, .p)` *keeps* all matching elements; `discard(.x, .p)` *drops* all matching elements.

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"))
detect(df, is.factor)
#> [1] a b c
#> Levels: a b c
detect_index(df, is.factor)
#> [1] 2

str(keep(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(discard(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
```

All of these functions could be implemented by first computing a logical vector, e.g. `map_lgl(.x, .p)`, and then computing on that. However, that is a little inefficient because you can often exit early. For example, in

9.4.3 `map_if()` and `modify_if()`

`map()` and `modify()` come in variants that also take predicate functions, transforming only the elements of `.x` with `.p` is `TRUE`.

```
str(map_if(iris, is.numeric, mean))
#> List of 5
```

```
#>  $ Sepal.Length: num 5.84
#>  $ Sepal.Width : num 3.06
#>  $ Petal.Length: num 3.76
#>  $ Petal.Width : num 1.2
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
str(modify_if(iris, is.numeric, mean))
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.84 5.84 5.84 5.84 5.84 ...
#>  $ Sepal.Width : num  3.06 3.06 3.06 3.06 3.06 ...
#>  $ Petal.Length: num  3.76 3.76 3.76 3.76 3.76 ...
#>  $ Petal.Width : num  1.2 1.2 1.2 1.2 1.2 ...
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
str(map(keep(iris, is.numeric), mean))
#> List of 4
#>  $ Sepal.Length: num 5.84
#>  $ Sepal.Width : num 3.06
#>  $ Petal.Length: num 3.76
#>  $ Petal.Width : num 1.2
```

9.4.4 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. What's the relationship between `which()` and `Position()`? What's the relationship between `where()` and `Filter()`?

3. 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.

4. 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.)

5. 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.

6. 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])
}
```

9.5 Base functionals

Base R functionals have more of a mathematical/statistical flavour.

9.5.1 Matrices 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`).

9.5.2 tapply()

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
#> 70.0 75.6
```

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] 71 70 68 70 70 72 75 71 69 64
#>
#> $B
#>  [1] 74 81 78 80 75 79 80 74 70 72 73 71
```

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
#> 70.0 75.6
```

Being able to rewrite `tapply()` as a combination of `split()` and `sapply()` is a good indication that we've identified some useful building blocks.

9.5.3 Mathmatical

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
```

9.5.4 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. 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.

10

Function factories

10.1 Introduction

A **function factory** is a function that makes functions. Here's a very simple example: we use a function factory (`power1()`) to make two child functions (`square()` and `cube()`):

```
power1 <- function(exp) {
  force(exp)

  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)
```

I'll call `square()` and `cube()` **manufactured functions**, but this is just a term to ease communication with other humans: from R's perspective they are no different to functions created any other way.

```
square(3)
#> [1] 9
cube(3)
#> [1] 27
```

You have already learned about the individual components that make function factories possible:

- In Section **??**, you learned about R's "first class" functions. In R, you bind a function to name in the same way as you bind any object to a name: with `<-`.

- In Section **??**, you learned that a function captures (encloses) the environment in which it is created.
- In Section **??**, you learned that a function creates a new execution environment every time it is run. This environment is usually ephemeral, but here it becomes the enclosing environment of the manufactured function.

In this chapter, you'll learn how the non-obvious combination of these three features lead to the function factory. You'll also see examples of their usage in visualisation and statistics.

Of the three main functional programming tools (functionals, function factories, and function operators), function factories are probably the least useful. Generally, they don't tend to reduce overall code complexity. Instead, they tend to partition complexity into more easily digested chunks. Function factories are also an important building block for the very useful function operators, which you'll learn about in Chapter @ref(#function-operators).

Outline

- Section **??** begins the chapter with an explanation of how function factories work, pulling together ideas from scoping and environments. You'll also see how function factories can be used to implement a "memory" for functions, allowing data to persist across calls.
- Section **??** illustrates the use of function factories with examples from ggplot2. You'll see two examples of how ggplot2 works with user supplied function factories, and one example of where ggplot2 uses a function factory internally.
- Section **??** uses function factories to tackle three challenges from statistics: understanding the Box-Cox transform, solving maximum likelihood problems, and drawing bootstrap resamples.
- Section **??** shows how you can combine function factories and functionals to rapidly generate a family of functions from data.

Prerequisites

Make sure you're familiar with the contents of Sections **??** (first class functions), **??** (function environments), and **??** (execution environments) mentioned above.

Function factories only need base R. We'll use a little rlang to peek inside of them more easily, and we'll use ggplot2 and scales to explore the use of function factories in visualisation.

```
# The development version includes some printing tweaks that we need here
# devtools::install_github("r-lib/rlang")
library(rlang)

library(ggplot2)
library(scales)
```

10.2 Factory fundamentals

The key idea that makes function factories work can be expressed very concisely:

> The enclosing environment of the manufactured function is an execution environment of the function factory.

It only takes few words to express these big ideas, but it takes a lot more work to really understand what this means. This section will help you put the pieces together with interactive exploration and some diagrams.

10.2.1 Environments

Let's start by taking a look at `square()` and `cube()`:

```
square
#> function(x) {
#>     x ^ exp
#>   }
#> <environment: 0x7fb371c999c8>

cube
#> function(x) {
#>     x ^ exp
#>   }
#> <bytecode: 0x7fb370cd1750>
#> <environment: 0x7fb3728746a0>
```

Printing manufactured functions is not revealing because the bodies are identical; its the contents of the enclosing environment that's important. We can get a little more insight by using `rlang::env_print()`. That shows us that we

have two different environments (each of which was originally an execution environment of `power()`). The environments have the same parent, which is the enclosing environment of `power1()`, the global environment.

```
env_print(square)
#> <environment: 0x7fb371c999c8>
#>   parent: <environment: global>
#>   bindings:
#>    * exp: <dbl>

env_print(cube)
#> <environment: 0x7fb3728746a0>
#>   parent: <environment: global>
#>   bindings:
#>    * exp: <dbl>
```

`env_print()` shows us that both environments have a binding to `exp`, but we want to see its value[1]. That's easily done with `env_get()`:

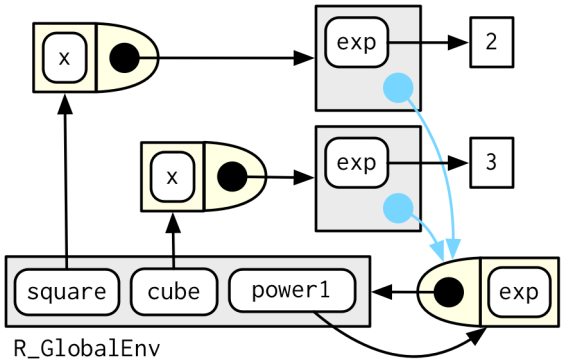
```
env_get(square, "exp")
#> [1] 2

env_get(cube, "exp")
#> [1] 3
```

This is what makes manufactured functions behave differently from one another: names in the enclosing environment are bound to different values.

10.2.2 Diagram conventions

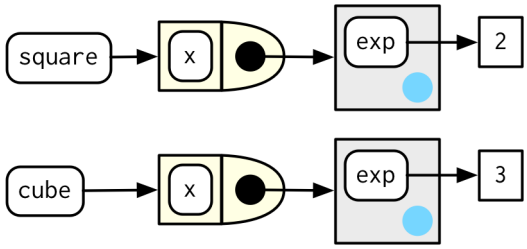
We can also show these relationships in a diagram:

[1] A future version of `env_print()` is likely to do better at summarising the contents so you don't need this step.



There's a lot going on this diagram and some of the details aren't that important. We can simplify considerably by using two conventions:

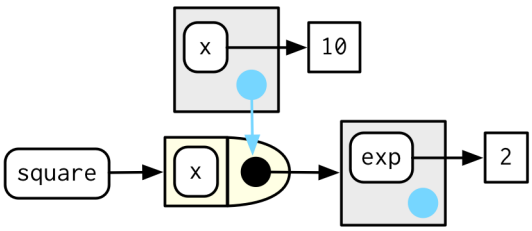
- Any free floating symbol lives in the global environment.
- Any environment without an explicit parent inherits from the global environment.



This view, which focuses on the environments, doesn't show any direct link between `cube()` and `square()`. That's because the link is the through the body of the function, which is identical for both, but is not shown in this diagram.

To finish up, let's look at the execution environment of `square(10)`. When `square()` executes `x ^ exp` it finds `x` in the execution environment and `exp` in its enclosing environment.

```
square(10)
#> [1] 100
```



10.2.3 Stateful functions

Function factories also allow you to maintain state across function invocations, which is generally hard to do because of the fresh start principle described in Section **??**.

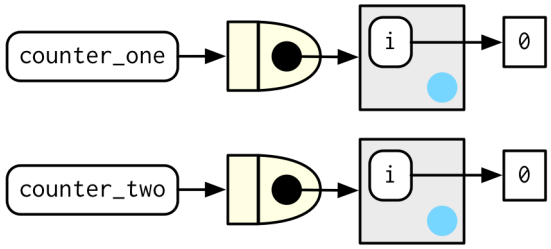
There are two things that make this possible:

- The enclosing environment of the manufactured function is unique and constant.
- R has a special assignment operator, <<-, which modifies bindings in the enclosing environment.

The usual assignment operator, <-, always creates a binding in the current environment. The **super assignment operator**, <<- rebinds an existing name found in a parent environment.

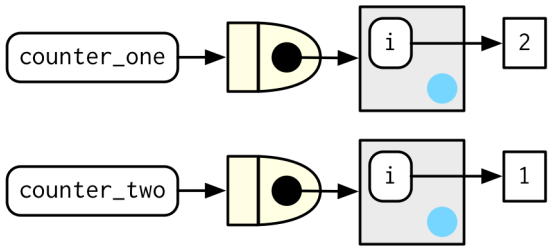
The following example example shows how we can combine these ideas to create a function that records how many times it has been called:

```
new_counter <- function() {
  i <- 0

  function() {
    i <<- i + 1
    i
  }
}

counter_one <- new_counter()
counter_two <- new_counter()
```



When the manufactured function is run `i <<- i + 1` will modify `i` in its enclosing environment. Because manufactured functions have independent enclosing environments, they have independent counts:

```
counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```



Stateful functions are best used in moderation. As soon as your function starts managing the state of multiple variables, it's better to switch to R6, the topic of Chapter **??**.

10.2.4 Potential pitfalls

There are two potential pitfalls to be aware of when creating your own function factories: forgetting to evaluate all inputs and accidentally capturing large objects.

Generally, you can rely on lazy evaluation to evaluate function inputs at the right time. However, there's a catch when it comes to function factories: if you don't eagerly evaluate every argument, it's possible to get confusing behaviour, as shown below.

```
power2 <- function(exp) {
  function(x) {
    x ^ exp
  }
}

exp2 <- 2
square2 <- power2(exp2)
exp2 <- 3

square2(2)
#> [1] 8
```

This is described in Section **??**, and happens when a binding changes in between calling the factory function and calling the manufactured function. This is likely to only happen rarely, but when it does, it will lead to a real head-scratcher of a bug. Avoid future pain by ensuring every argument is evaluated, using `force()` if the argument is only used by the manufactured function.

With most functions, you can rely on the GC to clean up any large temporary objects created inside a function. However, manufactured functions hold on to the execution environment, so you'll need to explicitly unbind any large temporary objects with `rm()`. Compare the sizes of `g1()` and `g2()` in the example below:

```
f1 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  function() m
}

g1 <- f1(1e6)
lobstr::obj_size(g1)
#> 8,013,648 B

f2 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  rm(x)
  function() m
}

g2 <- f2(1e6)
```

```
lobstr::obj_size(g2)
#> 13,488 B
```

10.2.5 Exercises

1. Base R contains two function factories, `approxfun()` and `ecdf()`. Read their documentation and experiment to figure out what the functions do and what they return.

2. 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`.

```
pick(1)(x)
# should be equivalent to
x[[1]]

lapply(mtcars, pick(5))
# should be equivalent to
lapply(mtcars, function(x) x[[5]])
```

3. Create a function that creates functions that compute the i^{th} 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))
```

4. What happens if you don't use a closure? Make predictions then, verify with the code below.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
```

5. What happens if you use <- instead of <<-? Make predictions, then verify with the code below.

```
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

10.3 Graphical factories

We'll begin our exploration of useful function factories with a few examples from ggplot2.

10.3.1 Labelling

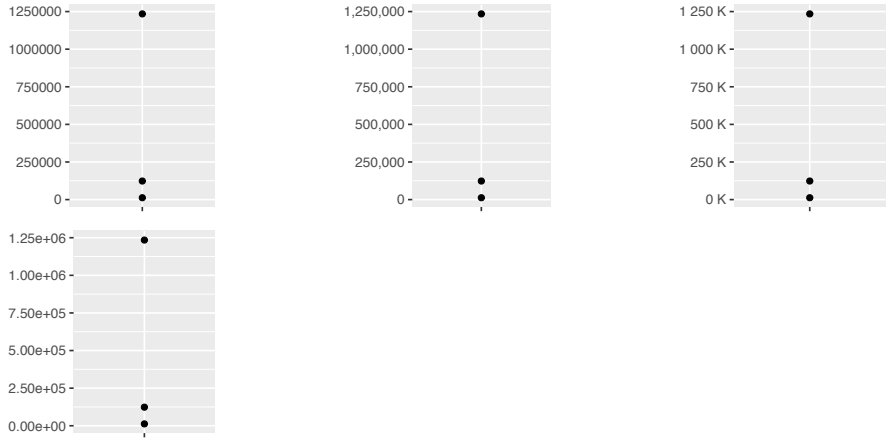
One of the goals of the scales (`http://scales.r-lib.org`) package is to make it easy to customise the labels on ggplot2. It provides many functions to control the fine details of axes and legends. One useful class of functions are the formatter functions[2] which which make it easier to control the appearance of axis breaks. The design of these functions might initially seem a little odd: they all return a function, which you have to call in order to format a number.

```
y <- c(12345, 123456, 1234567)
comma_format()(y)
#> [1] "12,345"    "123,456"   "1,234,567"

number_format(scale = 1e-3, suffix = " K")(y)
#> [1] "12 K"    "123 K"   "1 235 K"
```

In other words, the primary interface is a function factory. At first glance, this seems to add extra complexity for little gain. But it enables a nice interaction with ggplot2's scales, because they accept functions in the `label` argument:

[2]It's an unfortunate accident of history that scales uses function suffixes instead of function prefixes. That's because it was written before I understood the autocomplete advantages to using common prefixes instead of common suffixes.

```
df <- data.frame(x = 1, y = y)
core <- ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_continuous(breaks = 1, labels = NULL) +
  labs(x = NULL, y = NULL)

core
core + scale_y_continuous(label = comma_format())
core + scale_y_continuous(label = number_format(scale = 1e-3, suffix = " K"))
core + scale_y_continuous(label = scientific_format())
```



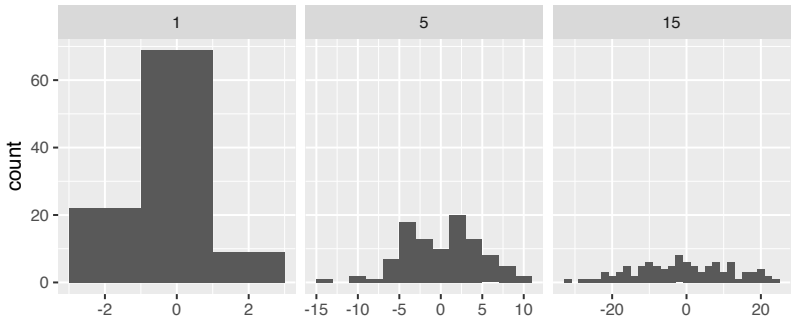
10.3.2 Histogram bins

A little known feature of `geom_histogram()` is that the `binwidth` argument can be a function. This is particularly useful because the function is executed once for each group, which means you can have different binwidths in different facets, which is otherwise not possible.

To illustrate this idea, and see where variable binwidth might be useful, I'm going to construct an example where a fixed binwidth isn't great.

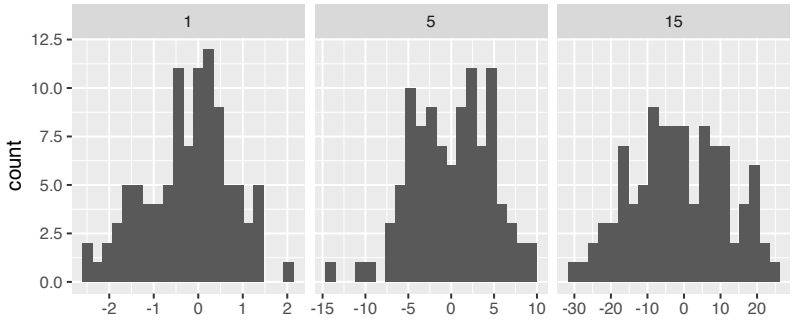
```
# construct some sample data with very different numbers in each cell
sd <- c(1, 5, 15)
n <- 100

df <- data.frame(x = rnorm(3 * n, sd = sd), sd = rep(sd, n))
```

```
ggplot(df, aes(x)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



Here each facet has the same number of observations, but the variability is very different. It would be nice if we could request that the binwidths vary so we get approximately the same number of observations in each bin. One way to do that is with a function factory that inputs the desired number of bins (`n`), and outputs a function that takes a numeric vector and returns a binwidth:

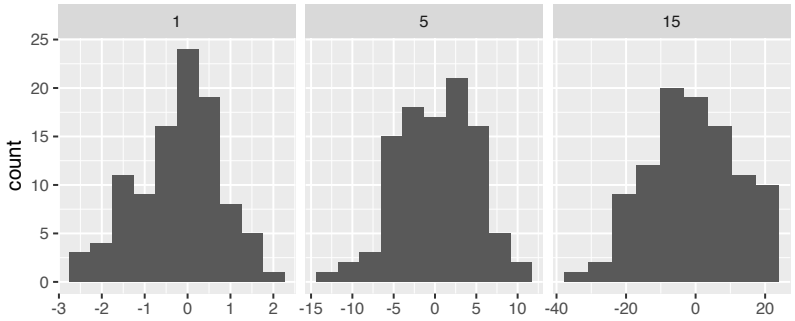
```
binwidth_bins <- function(n) {
  force(n)

  function(x) {
    (max(x) - min(x)) / n
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = binwidth_bins(20)) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



We could use this same pattern to wrap around the base R functions that automatically find the "optimal"[3] binwidth, `nclass.Sturges()`, `nclass.scott()`, and `nclass.FD()`:

```
base_bins <- function(type) {
  fun <- switch(type,
    Sturges = nclass.Sturges,
    scott = nclass.scott,
    FD = nclass.FD,
    stop("Unknown type", call. = FALSE)
  )

  function(x) {
    (max(x) - min(x)) / fun(x)
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = base_bins("FD")) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```

[3] ggplot2 doesn't expose these functions directly because I don't think the defintion of optimality needed to make the problem mathematically tractable is a good match to the actual needs of data exploration.



10.3.3 ggsave

Finally, I want to show a function factory used internally by ggplot2. `ggplot2:::plot_dev()` is used by `ggsave()` to go from a file extension (e.g. `png`, `jpeg` etc) to a graphics device function (e.g. `png()`, `jpeg()`). The challenge here arises because the base graphics devices have some minor inconsistencies which we need to paper over:

- Most have `filename` as first argument but some have `file`.
- The `width` and `height` of raster graphic devices use pixels units by default, but the vector graphics use inches.

A mildly simplified version of `plot_dev()` is shown below:

```
plot_dev <- function(ext, dpi = 96) {
  force(dpi)

  switch(ext,
    eps =  ,
    ps  =  function(filename, ...) {
      grDevices::postscript(
        file = filename, ..., onefile = FALSE,
        horizontal = FALSE, paper = "special"
      )
    },
    tex =  function(filename, ...) grDevices::pictex(file = filename, ...),
    pdf =  function(filename, ...) grDevices::pdf(file = filename, ...),
    svg  = function(filename, ...) svglite::svglite(file = filename, ...),
    emf  = ,
    wmf  = function(...) grDevices::win.metafile(...),
    png  = function(...) grDevices::png(..., res = dpi, units = "in"),
    jpg  = ,
```

```
    jpeg = function(...) grDevices::jpeg(..., res = dpi, units = "in"),
    bmp  = function(...) grDevices::bmp(..., res = dpi, units = "in"),
    tiff = function(...) grDevices::tiff(..., res = dpi, units = "in"),
    stop("Unknown graphics extension: ", ext, call. = FALSE)
  )
}

plot_dev("pdf")
#> function(filename, ...) grDevices::pdf(file = filename, ...)
#> <bytecode: 0x7fb373710190>
#> <environment: 0x7fb3728b10d0>
plot_dev("png")
#> function(...) grDevices::png(..., res = dpi, units = "in")
#> <bytecode: 0x7fb373a550d0>
#> <environment: 0x7fb370fa7c70>
```

10.3.4 Exercises

1. Compare and contrast `ggplot2::label_bquote()` with `scales::number_format()`

10.4 Statistical factories

More motivating examples for function factories come from statistics:

- The Box-Cox transformation.
- Bootstrap resampling.
- Maximum likelihood estimation.

All of these examples can be tackled without function factories, but I think function factories are a good fit for these problems and provide elegant solutions. These examples expect some statistical background, so feel free to skip if they don't make much sense to you.

10.4.1 Box-Cox transformation

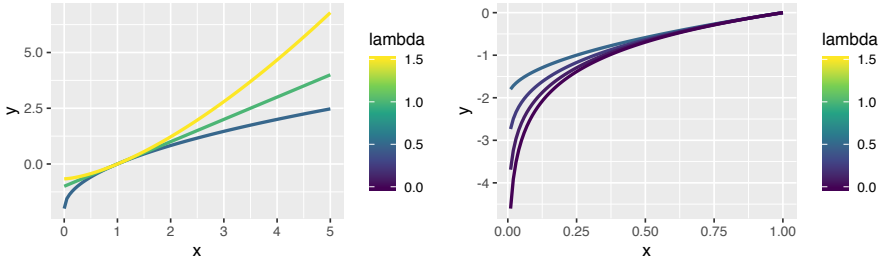
The Box-Cox transformation is a flexible transformation often used to transform data towards normality. It has a single parameter, λ which controls

the strength of the transformation. We could express the transformation as a simple two argument function:

```
boxcox1 <- function(x, lambda) {
  stopifnot(length(lambda) != 1)

  if (lambda == 0) {
    log(x)
  } else {
    (x ^ lambda - 1) / lambda
  }
}
```

But re-formulating as a function factory makes it easy to explore its behaviour with `stat_function()`:

```
boxcox2 <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}

stat_boxcox <- function(lambda) {
  stat_function(aes(colour = lambda), fun = boxcox2(lambda), size = 1)
}

ggplot(data.frame(x = c(0, 5)), aes(x)) +
  lapply(c(0.5, 1, 1.5), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))

# visually, log() does seem to make sense as the limit as lambda -> 0
ggplot(data.frame(x = c(0.01, 1)), aes(x)) +
  lapply(c(0.5, 0.25, 0.1, 0), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
```



In general, this allows you to use a Box-Cox transformation with any function that accepts a unary transformation function: you don't have to worry about that function providing ... to pass along additional arguments. I also think that the partitioning of `lambda` and `x` into two different function arguments is natural since `lambda` plays quite a different role than `x`.

10.4.2 Bootstrap generators

Function factories are a useful approach for bootstrapping. Instead of thinking about a single bootstrap (you always need more than one!), you can think about a bootstrap **generator**, a function that yields a fresh bootstrap every time it is called:

```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    df[[var]][sample(n, n, replace = TRUE)]
  }
}

boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
#> [1] 18.1 22.8 21.5 14.7 21.4 17.3
head(boot_mtcars1())
#> [1] 19.2 19.2 14.3 21.0 13.3 21.4
```

The advantage of a function factory is more clear with a parametric bootstrap where we have to first fit a model. We can do this setup step once, when the factory is called, rather than once every time we generate the bootstrap:

```
boot_model <- function(df, formula) {
  mod <- lm(formula, data = df)
  fitted <- unname(fitted(mod))
  resid <- unname(resid(mod))
  rm(mod)

  function() {
    fitted + sample(resid)
  }
}

boot_mtcars2 <- boot_model(mtcars, mpg ~ wt)
head(boot_mtcars2())
#> [1] 23.1 24.3 23.0 19.1 19.1 16.2
head(boot_mtcars2())
#> [1] 30.2 17.4 31.3 26.1 17.8 16.7
```

I use `rm(mod)` because linear model objects are quite large (they include complete copies of the model matrix and input data) and I want to keep the manufactured function as small as possible.

10.4.3 Maximum likelihood estimation

The goal of maximum likelihood estimation (MLE) is to find the parameter values for a distribution that make the observed data "most likely". To do MLE, you start with a probability function. For example, take the Poisson distribution. If we know λ, we can compute the probability of getting a vector $\mathbf{x}$ of values (x_1, x_2, …, x_n) by multiplying the Poisson probability function as follows:

$$P(\lambda, \mathbf{x}) = \prod_{i=1}^{n} \frac{\lambda^{x_i} e^{-\lambda}}{x_i!}$$

In statistics, we almost always work with the log of this function. The log is a monotonic transformation which preserves important properties (i.e. the extrema occur in the same place), but has specific advantages:

- The log turns a product into a sum, which is easier to work with.
- Multiplying small numbers yields even smaller numbers, which makes the floating point approximation used by a computer less accurate.

Let's apply a log transformation to this probability function and simplify it as much as possible: